Computer Architecture

Prof. Dr. Nizamettin AYDIN

naydin@yildiz.edu.tr

http://www.yildiz.edu.tr/~naydin

The Von Neumann Model/Architecture

- Also called stored program computer (instructions in memory).
- Two key properties:
  - Stored program
    - Instructions stored in a linear memory array
    - Memory is unified between instructions and data
      - The interpretation of a stored value depends on the control signals
        - When is a value interpreted as an instruction?
  - Sequential instruction processing
    - One instruction processed (fetched, executed, and completed) at a time

A computing System (Reminder)

- What is a computer?
  - in terms of what?
    - Functional
    - Structural

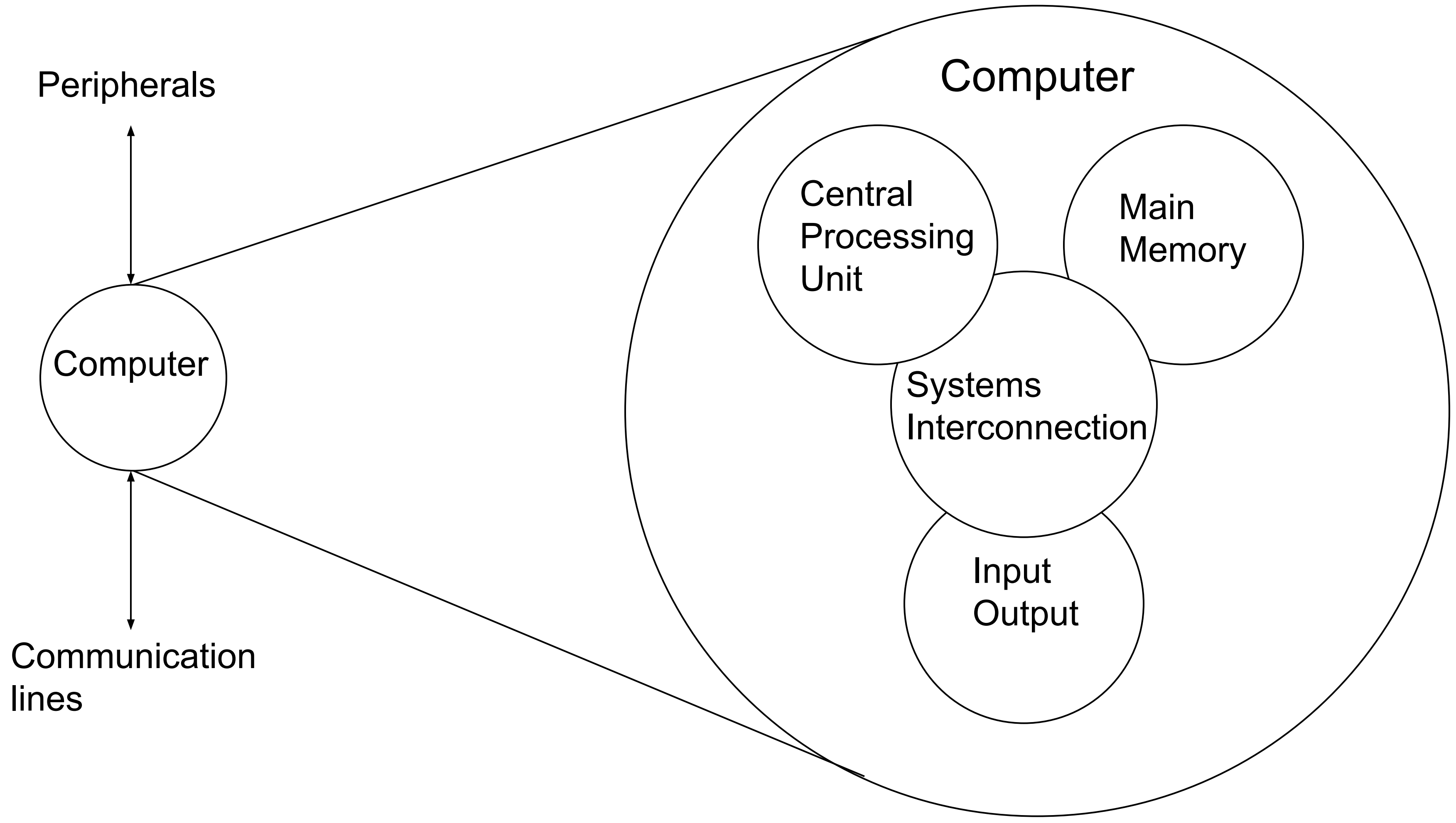
Functional definition

- Data processing
- Data storage
- Data movement
- Control

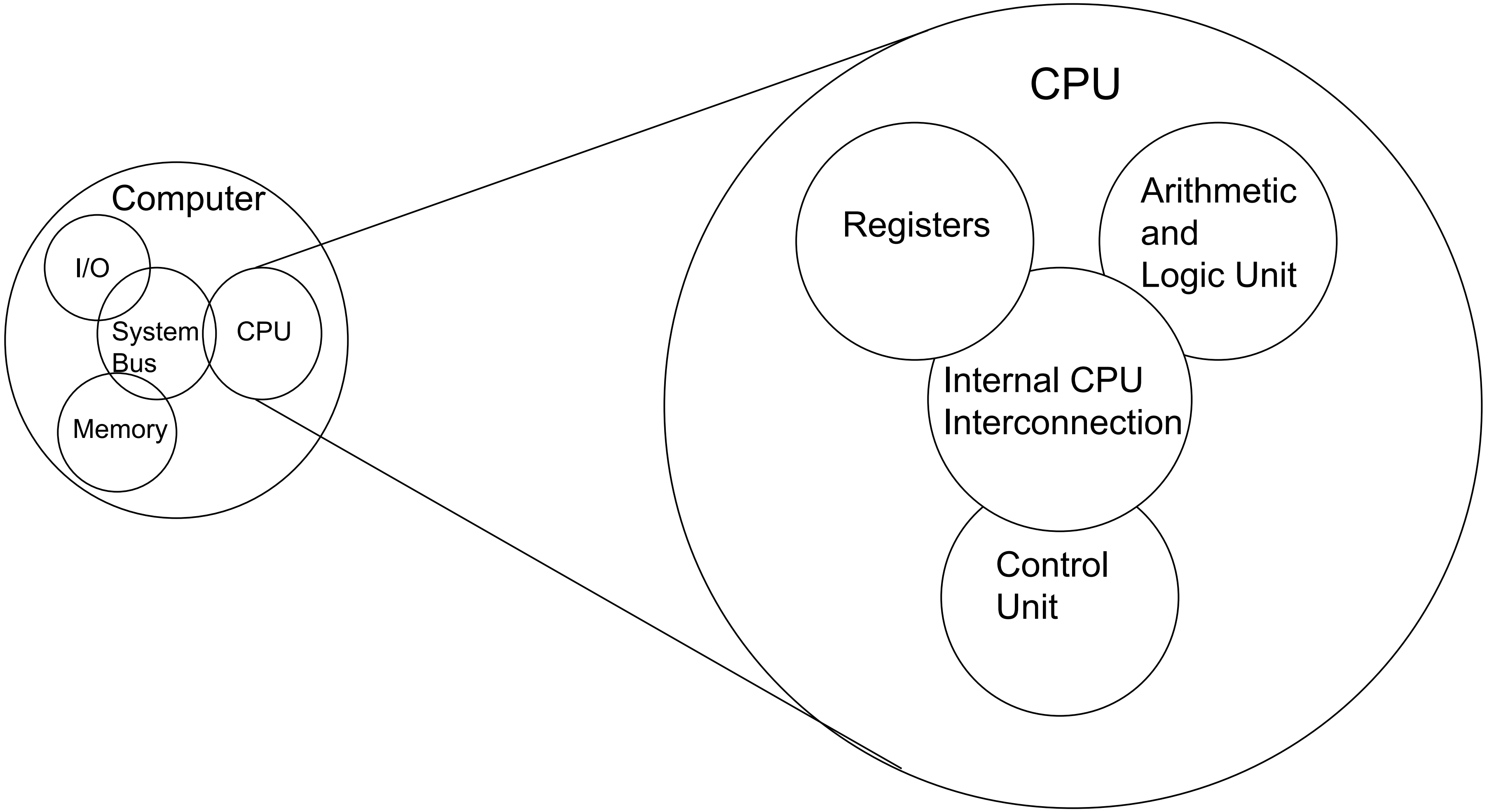
Structural definition

- Central processing unit
- Main memory
- Input/Output
- System interconnection

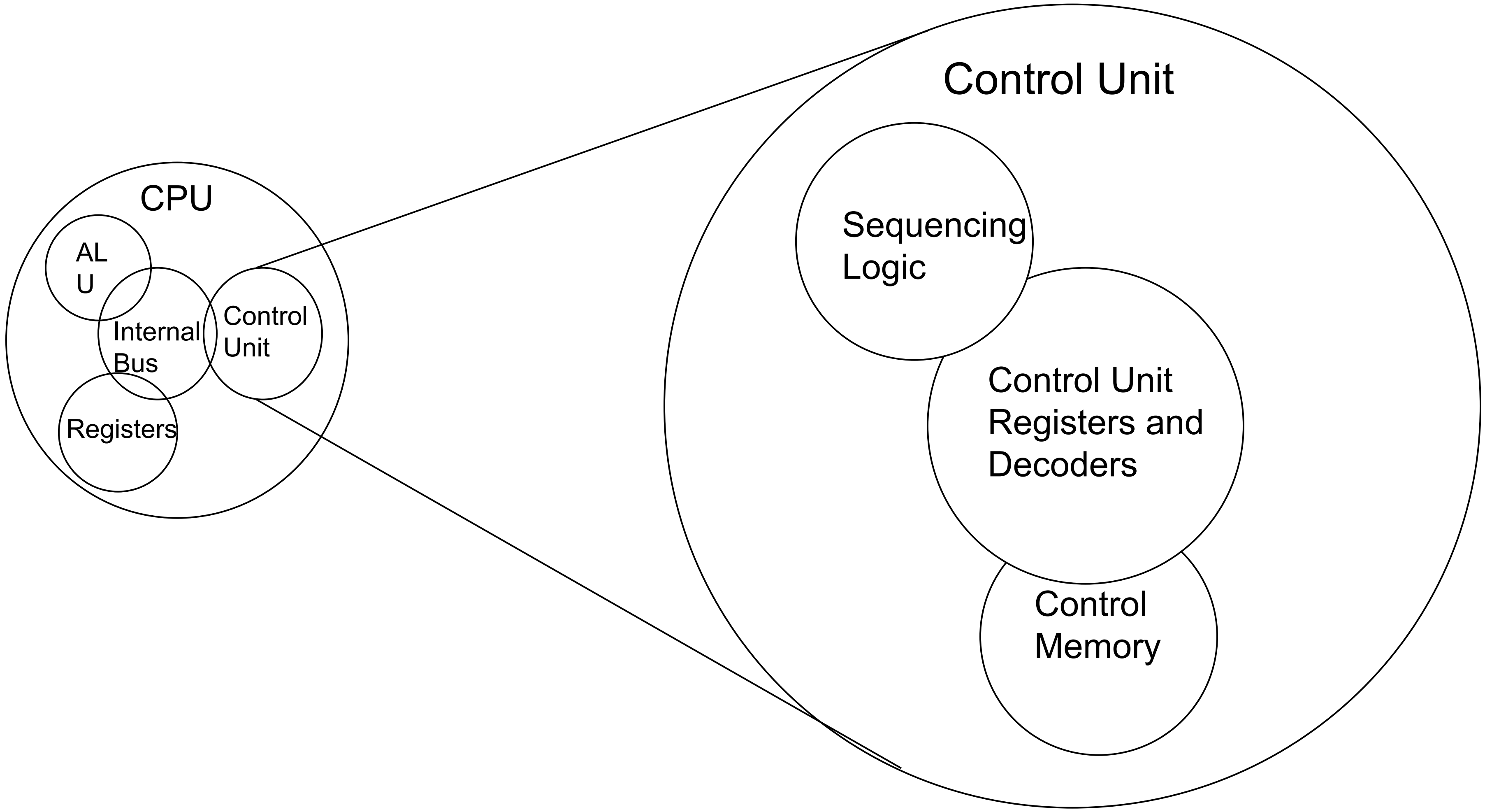
Structure - Top Level



Structure - The CPU



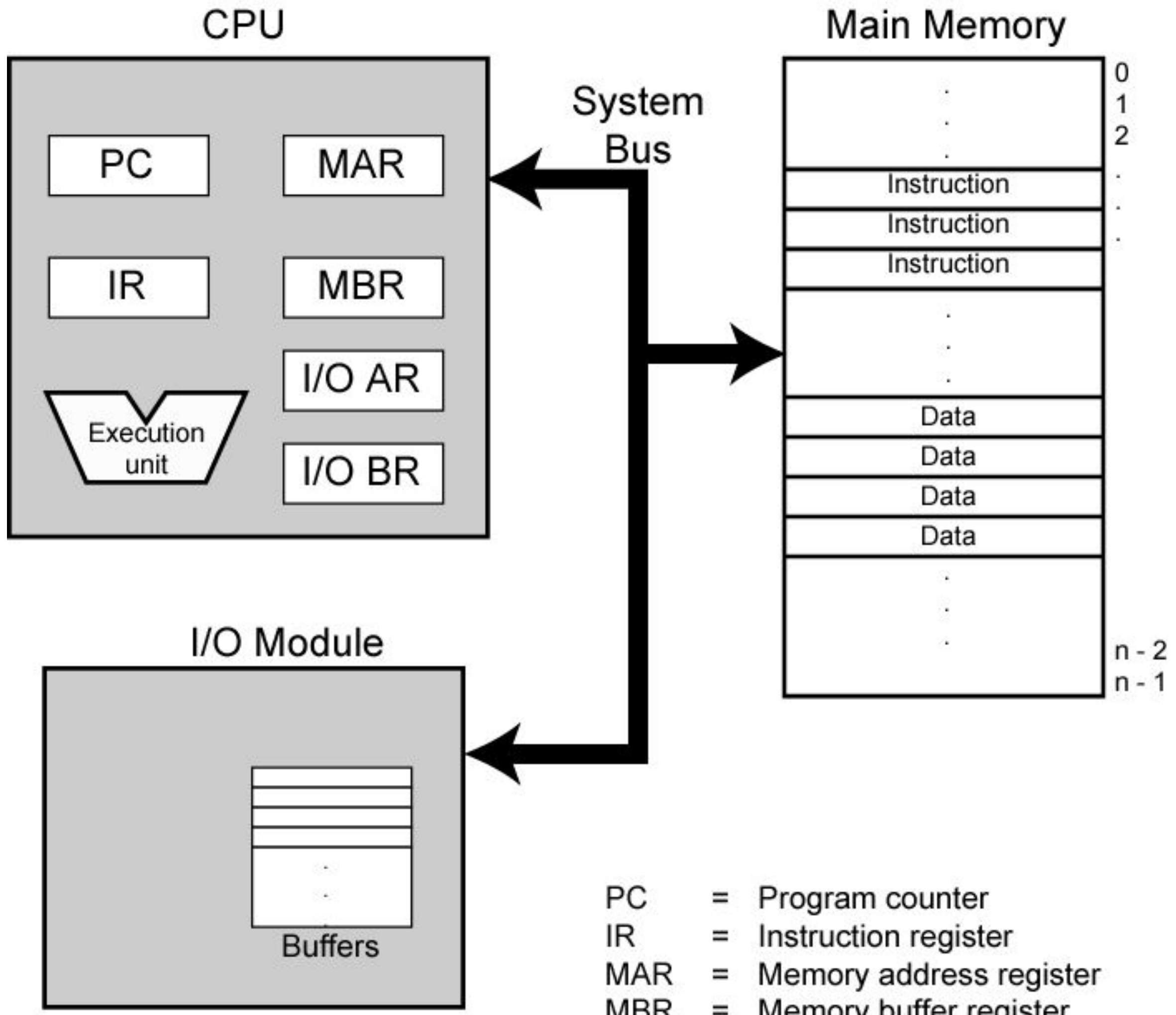
Structure - The Control Unit



The Stored Program Computer

- 1943: ENIAC
  - Presper Eckert and John Mauchly -- first general electronic computer. **(or was it John V. Atanasoff in 1939?)**
  - Hard-wired program -- settings of dials and switches.
- 1944: Beginnings of EDVAC
  - among other improvements, includes program stored in memory
- 1945: John von Neumann
  - wrote a report on the stored program concept, known as the *First Draft of a Report on EDVAC*
- The basic structure proposed in the draft became known as the "von Neumann machine" (or model).
  - a *memory*, containing instructions and data
  - a *processing unit*, for performing arithmetic and logical operations
  - a *control unit*, for interpreting instructions

Von Neumann Model



PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

ENIAC - details

http://www.seas.upenn.edu/~museum/

- Decimal (not binary)
- 20 accumulators of 10 digits
- Programmed manually by switches
- 18,000 vacuum tubes
- 30 tons
- 15,000 square feet
- 140 kW power consumption
- 5,000 additions per second

Interface to Memory

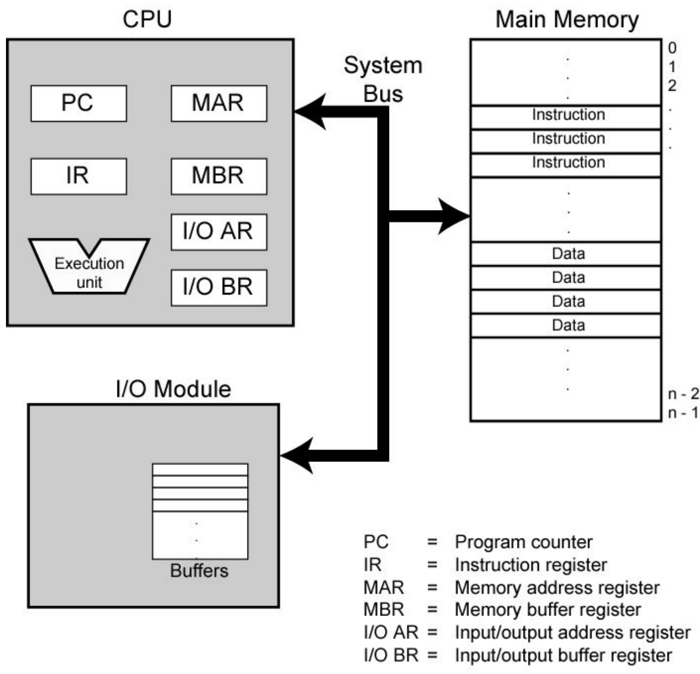
- How does processing unit get data to/from memory?
- MAR: Memory Address Register
- MBR (MDR): Memory Buffer (Data) Register

To LOAD a memory location (A):

1. Write the address (A) into the MAR.
2. Send a “read” signal to the memory.
3. Read the data from MBR.

To STORE a value (X) to a location (A):

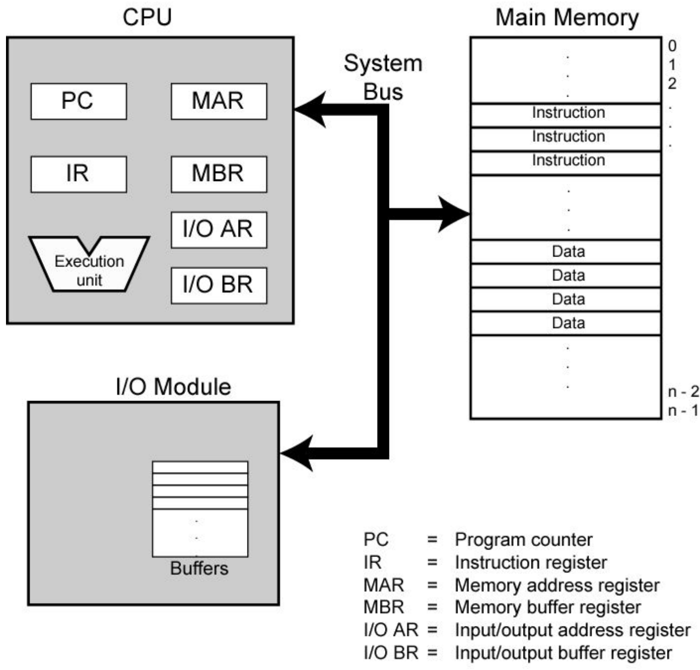
4. Write the data (X) to the MBR.
5. Write the address (A) into the MAR.
6. Send a “write” signal to the memory.



PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

Processing Unit

- Functional Units (Execution unit)
  - – ALU = Arithmetic and Logic Unit
  - – could have many functional units. some of them special-purpose (multiply, square root, …)
- Registers
  - – Small, temporary storage
  - – Operands and results of functional units
- Word Size
  - – number of bits normally processed by ALU in one instruction
  - – also width of registers



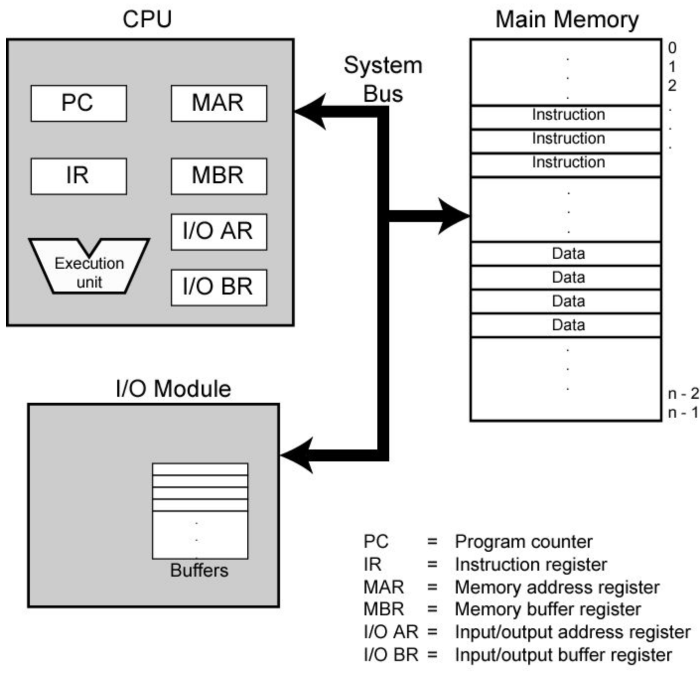
Input and Output

- Devices for getting data into and out of computer memory

- Each device has its own interface, usually a set of registers (I/OAR and I/OBR)

- Some devices provide both input and output
  - <span style="color:red">disk, network</span>

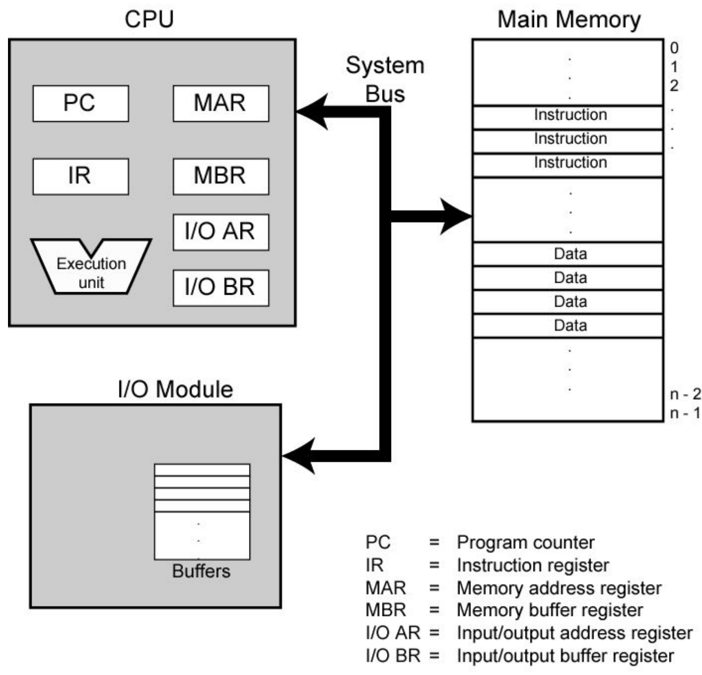
- Program that controls access to a device is usually called a *driver*.



Control Unit

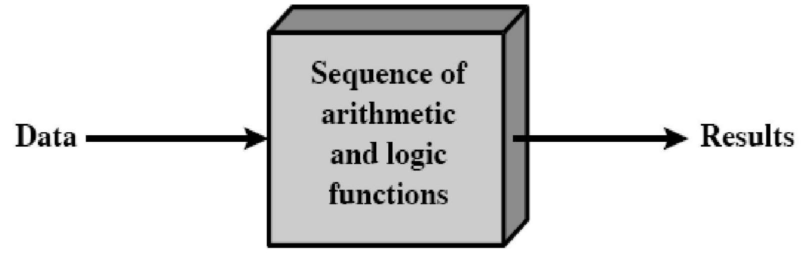
- Orchestrates execution of the program

- Instruction Register (IR) contains the *current instruction*.
- Program Counter (PC) contains the *address* of the next instruction to be executed.
- Control unit:
  - reads an instruction from memory
    - the instruction's address is in the PC
  - interprets the instruction, generating signals that tell the other components what to do
    - an instruction may take many *machine cycles* to complete

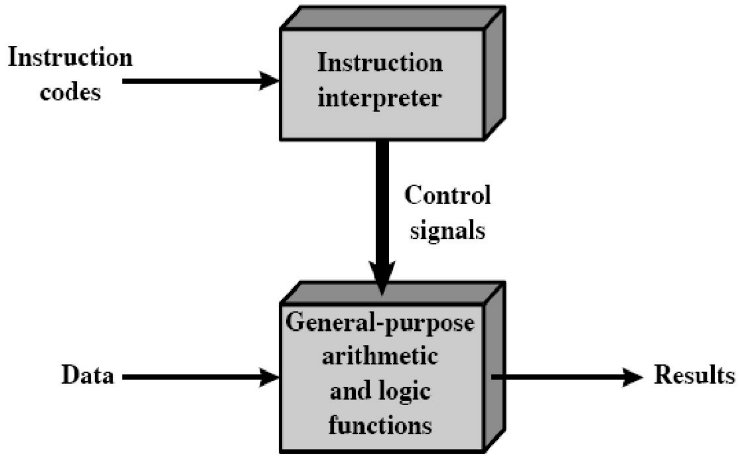


Program Concept

- Hardwired systems are inflexible
- General purpose hardware can do different tasks, given correct control signals
- Instead of re-wiring, supply a new set of control signals



(a) Programming in hardware

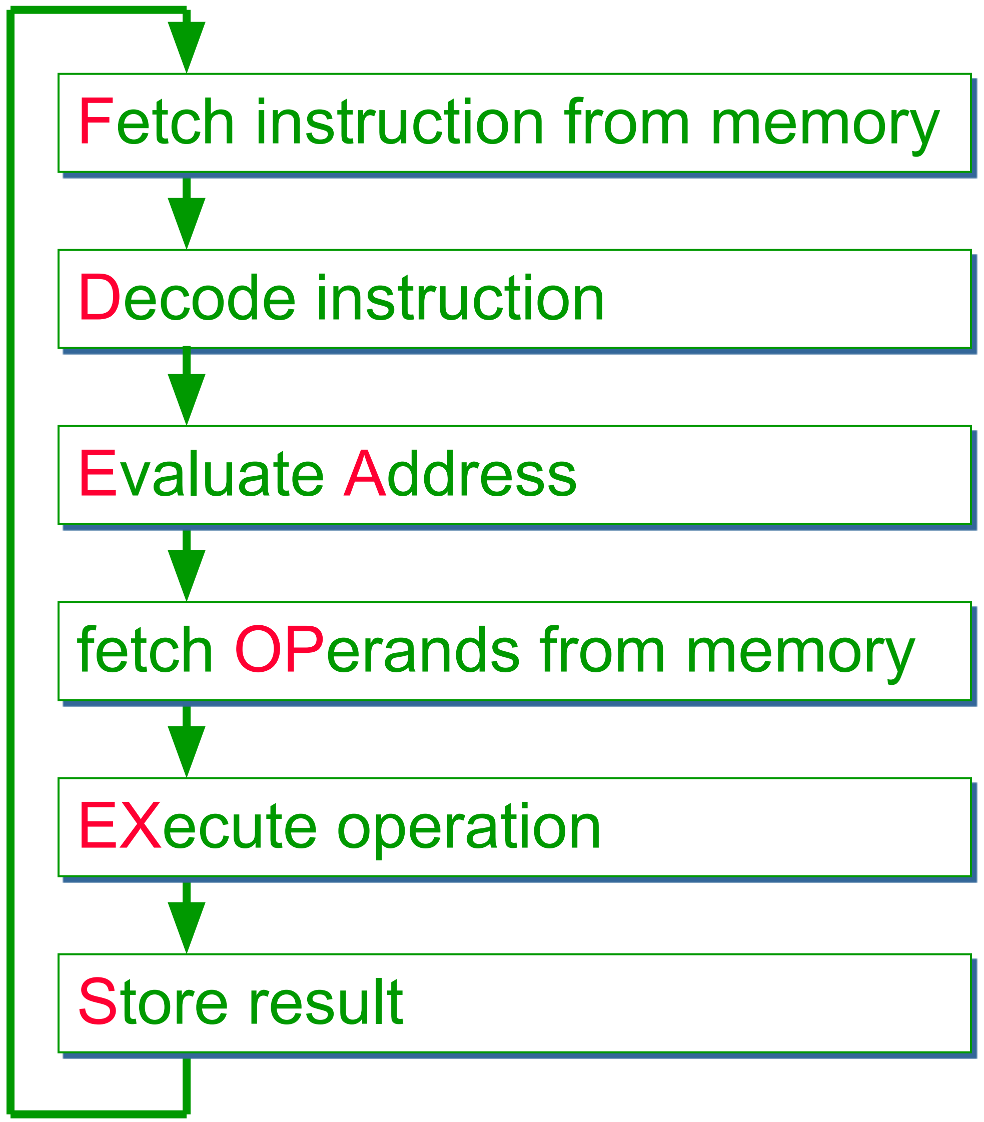


(b) Programming in software

What is a program?

- A sequence of steps
- For each step, an arithmetic or logical operation is done
- For each operation, a different set of control signals is needed

Instruction Processing



Instruction

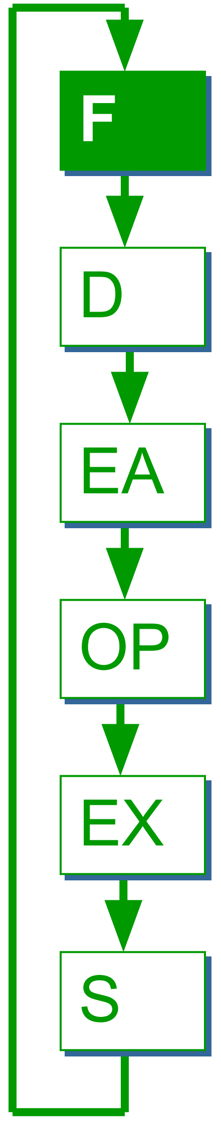
- The instruction is the fundamental unit of work.
- Specifies two things:
  - *opcode*: operation to be performed
  - *operands*: data/locations to be used for operation
- An instruction is encoded as a sequence of bits.
  (*Just like data!*)
  - **Often, but not always, instructions have a fixed length, such as 16 or 32 bits.**
  - **Control unit interprets instruction: generates sequence of control signals to carry out operation.**
  - **Operation is either executed completely, or not at all.**

- A computer's instructions and their formats is known as its ***Instruction Set Architecture* (*ISA*)**.

Instruction Processing: FETCH

- Load next instruction (at address stored in PC) from memory into Instruction Register (IR).
  - Copy contents of PC into MAR.
  - Send "read" signal to memory.
  - Copy contents of MBR into IR.

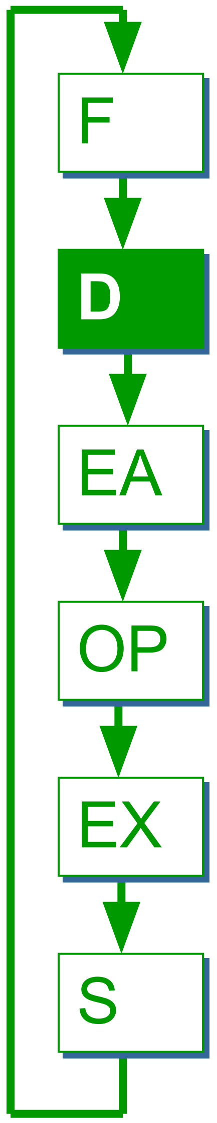
- Then increment PC, so that it points to the next instruction in sequence.
  - PC becomes PC+1.



Instruction Processing: DECODE

- First identify the opcode.
  - A n-to-2^n decoder asserts a control line corresponding to the desired opcode.

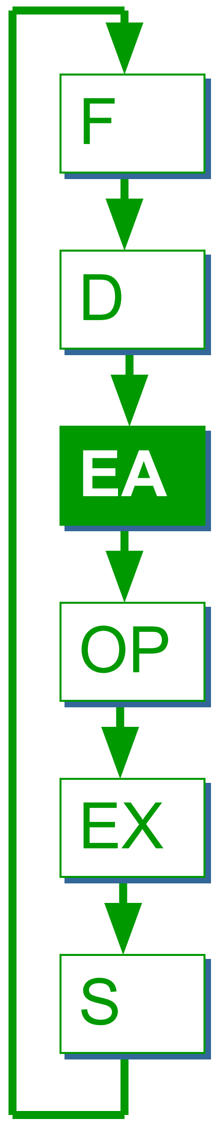
- Depending on opcode, identify other operands from the remaining bits.



Instruction Processing: EVALUATE ADDRESS

- For instructions that require memory access, compute address used for access.

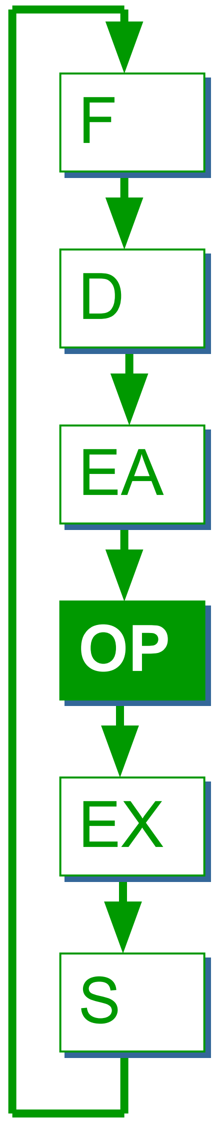
- Examples:
  - add offset to base register
  - add offset to PC
  - add offset to zero



Instruction Processing: FETCH OPERANDS

- Obtain source operands needed to perform operation.

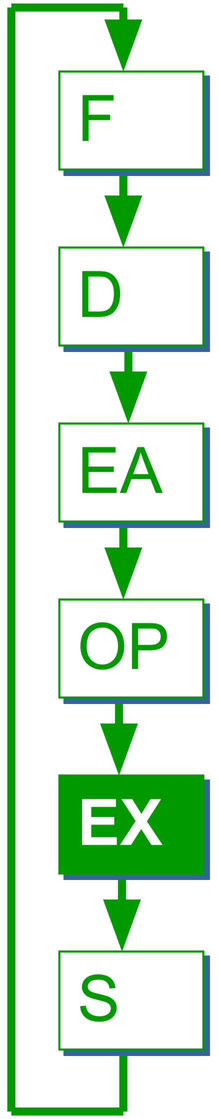
- Examples:
  - load data from memory (LDA)
  - read data from register file (ADD)



Instruction Processing: EXECUTE

- Perform the operation,
  using the source operands.

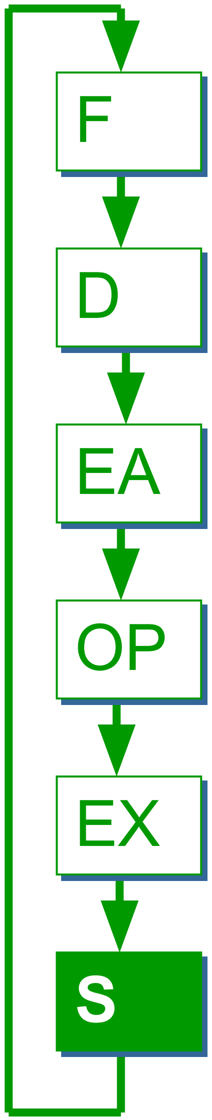
- Examples:
  - send operands to ALU and assert ADD signal
  - do nothing (e.g., for loads and stores)



Instruction Processing: STORE RESULT

- Write results to destination. (register or memory)

- Examples:
  - result of ADD is placed in destination register
  - result of memory load is placed in destination register
  - for store instruction, data is stored to memory
    - write address to MAR, data to MBR
    - assert WRITE signal to memory



Changing the Sequence of Instructions

- In the FETCH phase, we increment the Program Counter by 1.

- What if we don't want to always execute the instruction that follows this one?
  - examples: loop, if-then, function call

- Need special instructions that change the contents of the PC.
- These are called *control instructions*.
  - jumps are unconditional -- they always change the PC
  - branches are conditional -- they change the PC only if some condition is true (e.g., the result of an ADD is zero)

Instruction Processing Summary

- Instructions look just like data -- it's all interpretation.

- Four basic kinds of instructions:
  - Data processing instructions
    - Arithmetic and logic instructions (ADD, AND, …)
  - Data storage instructions
    - Memory instructions (LDA, STA)
  - Data movement instructions
    - I/O instructions (IN, OUT, …)
  - Program flow control instructions
    - Test and branch instructions (JMP, BRP, …)

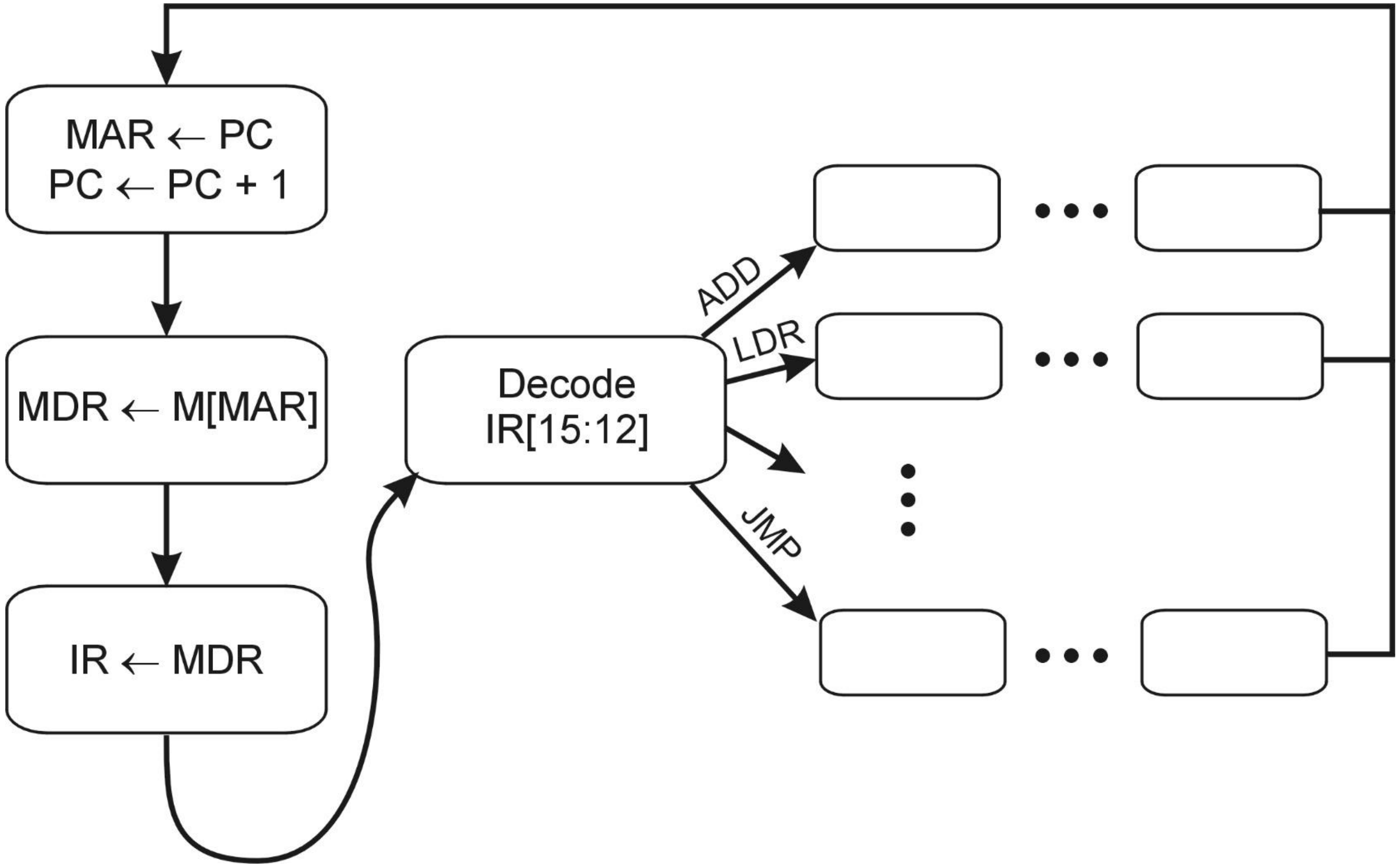
- Six basic phases of instruction processing:

  F → D → EA → OP → EX → S

  - not all phases are needed by every instruction
  - phases may take variable number of machine cycles

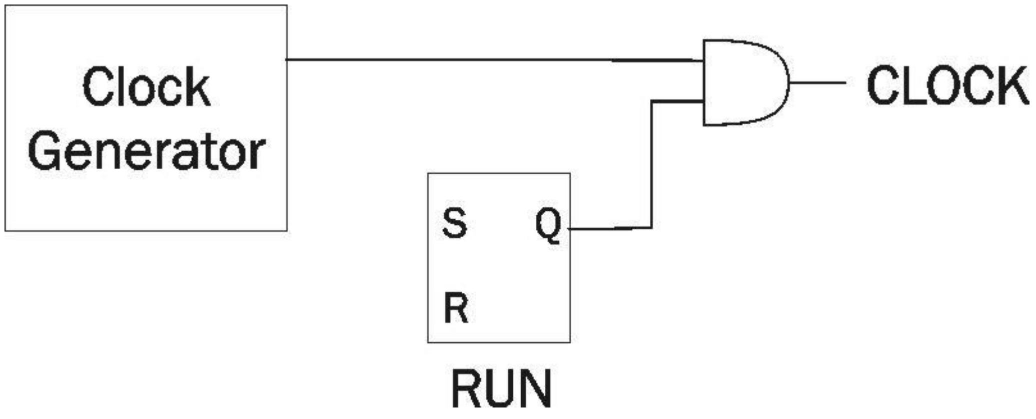
Control Unit State Diagram

- The control unit is a state machine.
- Here is part of a simplified state diagram:



Stopping the Clock

- Control unit will repeat instruction processing sequence as long as clock is running.
  - **If not processing instructions from your application, then it is processing instructions from the Operating System (OS).**
  - **The OS is a special program that manages processor and other resources.**
- To stop the computer:
  - **AND the clock generator signal with ZERO**
  - **When control unit stops seeing the CLOCK signal, it stops processing.**



Dataflow Model of a Computer

Von Neumann model

- An instruction is fetched and executed in control flow order
  - As specified by the instruction pointer
  - Sequential unless explicit control flow instructio

Dataflow model

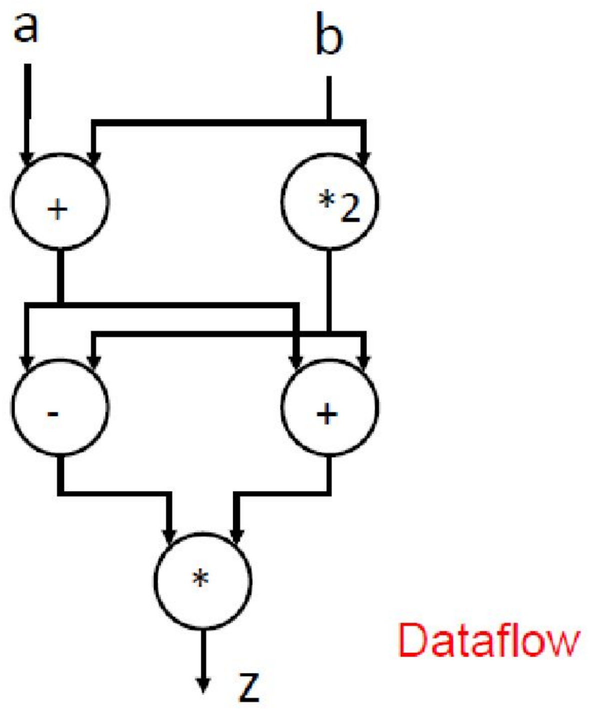
- An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no instruction pointer
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies "who" should receive the result
    - An instruction can "fire" whenever all operands are received
  - Potentially many instructions can execute at the same time
  - Inherently more parallel

von Neumann vs Dataflow

- Consider a von Neumann program (sequential) vs dataflow execution

```
v <= a + b;
w <= b * 2;
x <= v - w
y <= v + w
z <= x * y
```

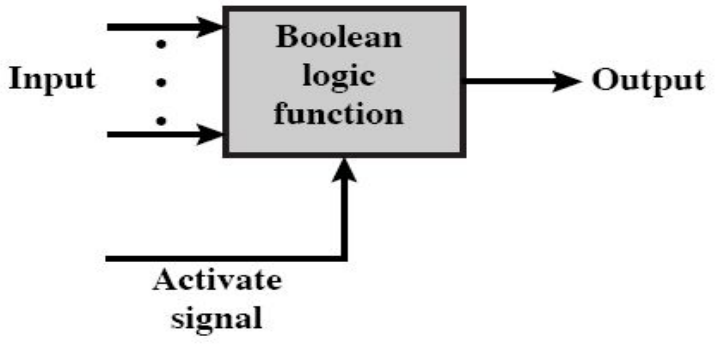
Sequential



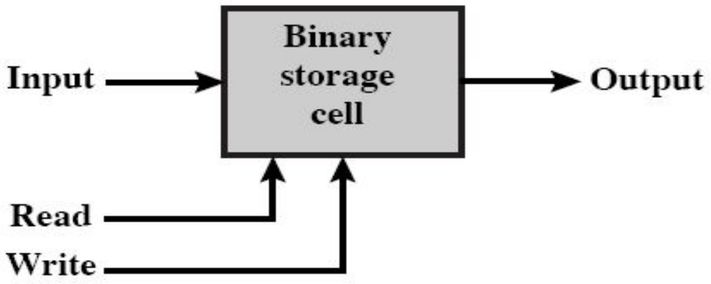
- Which model is more natural to you as a programmer?
- All major instruction set architectures today use von Neumann
  - x86, ARM, MIPS, SPARC, Alpha, POWER PC

Fundamental Computer Elements

Fundamental computer elements



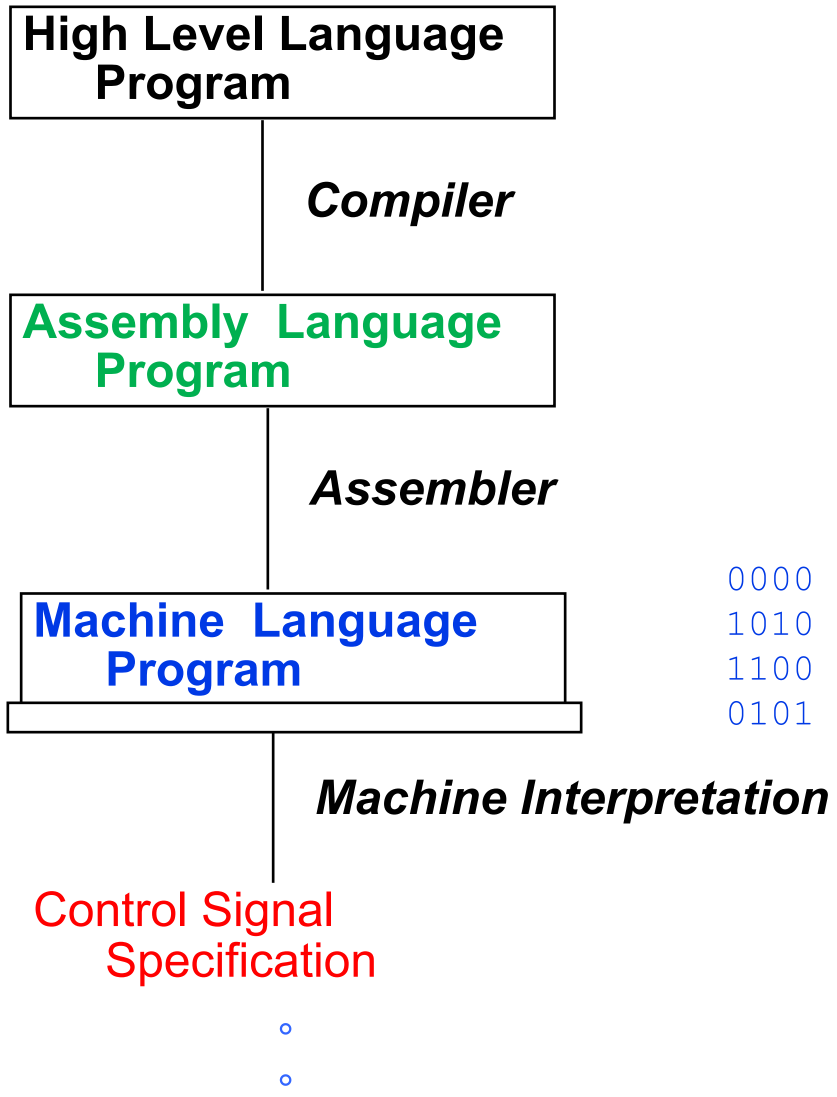
(a) Gate



(b) Memory cell

- Data processing
  - Provided by gates
- Data storage
  - Provided by memory cells
- Data movement
  - The paths between components are used to move data from/to memory
- Control
  - The paths between components can carry control signals

Levels of Representation


High Level Language Program
Compiler
Assembly Language Program
Assembler
Machine Language Program
Machine Interpretation
Control Signal Specification
°
°

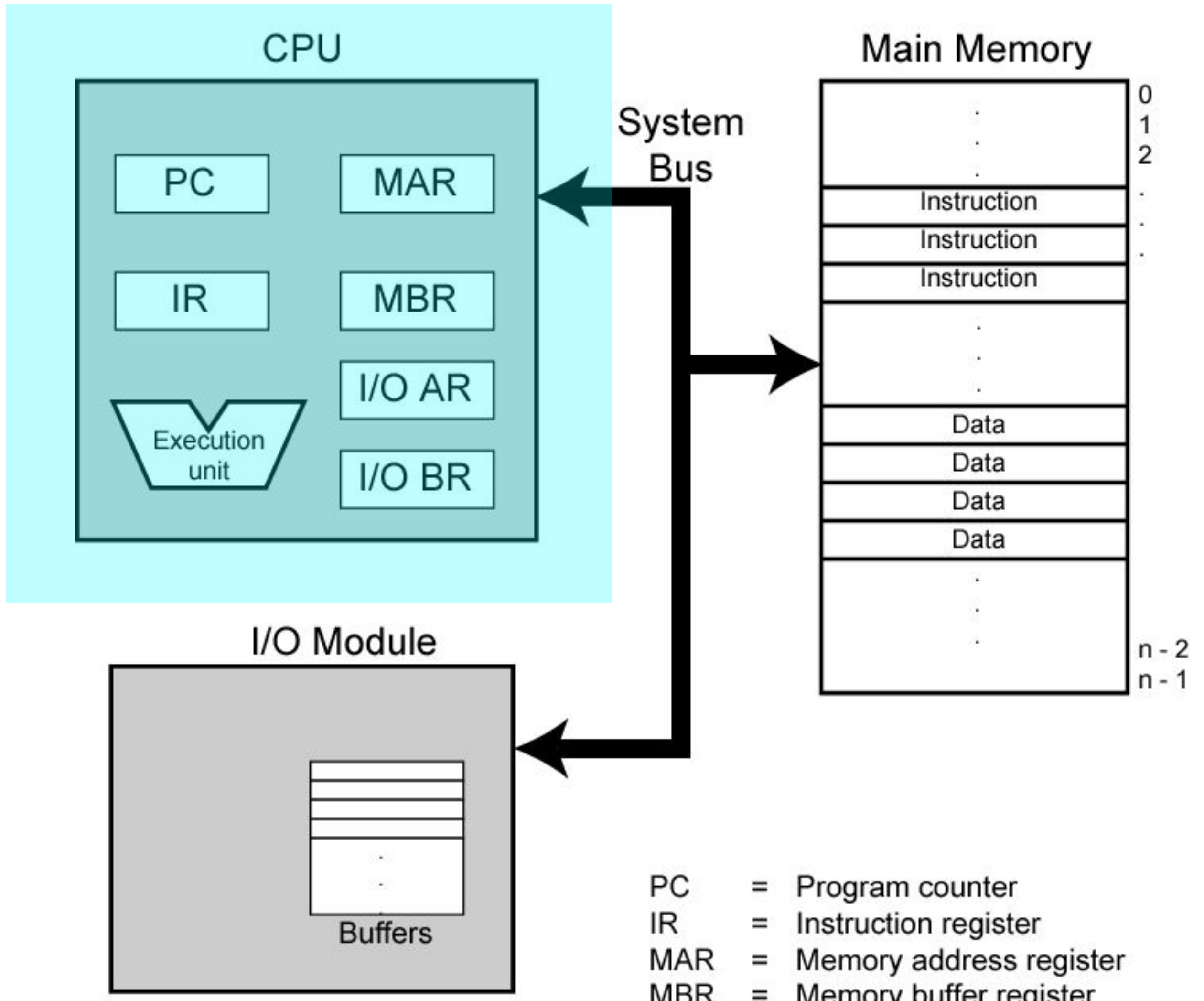
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $15, 0($2)
lw   $16, 4($2)
sw   $16, 0($2)
sw   $15, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

ALUOP[0:3] <= InstReg[9:11] & MASK

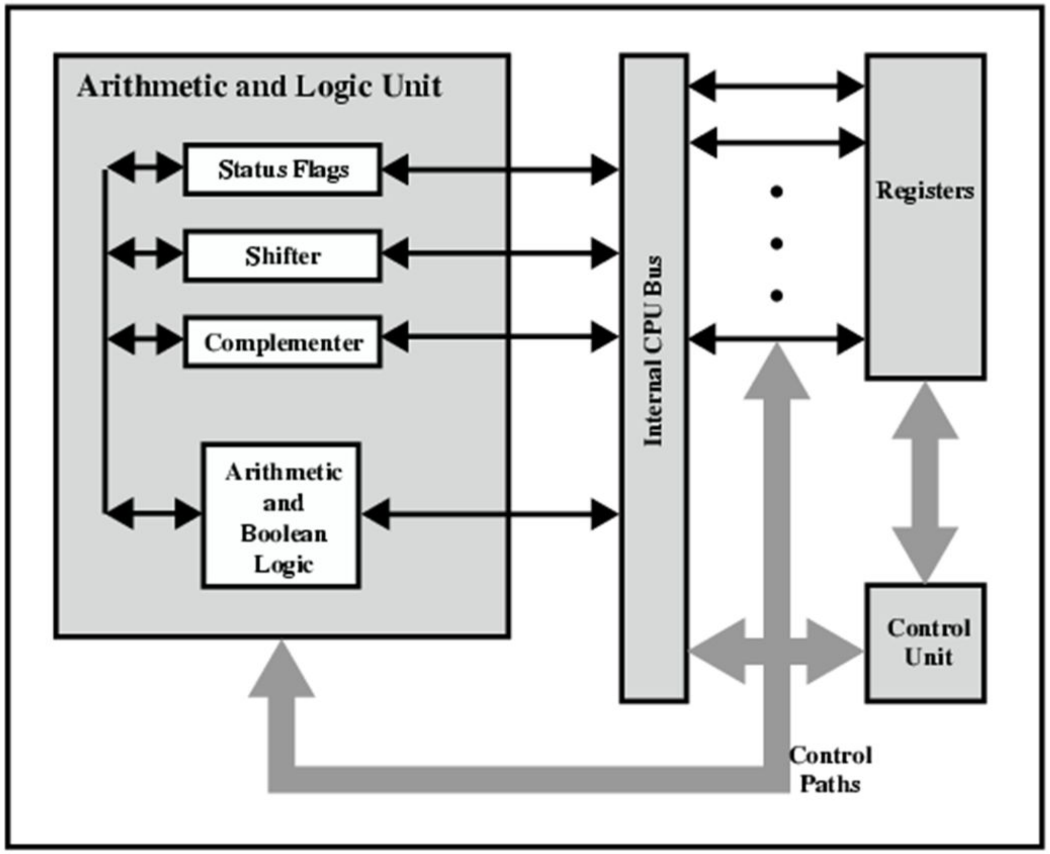
Computer Components-CPU



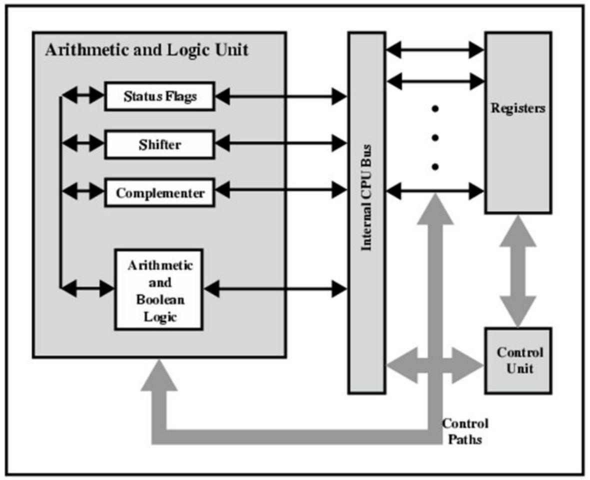
PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

CPU Structure

- CPU must:
  - Fetch instructions
  - Interpret instructions
  - Fetch data
  - Process data
  - Write data



Registers



- CPU must have some working space (temporary storage)
  - Called registers
- Number and function vary between processor designs
- One of the major design decisions
- Top level of memory hierarchy

Registers in the µP perform two roles:

- User-visible registers
  - Enable the machine- or assembly language programmer to minimize main memory references by optimizing use of registers

- Control and status registers
  - Used by the control unit to control the operation of the processor and by priviliged, operating system programs to control the execution of programs

User Visible Registers

- General Purpose registers
- Data registers
- Address registers
- Condition Codes (flags)

Condition Code Registers

- Sets of individual bits
  - e.g. result of last operation was zero
- Can be read (implicitly) by programs
  - e.g. Jump if zero
- Can not (usually) be set by programs

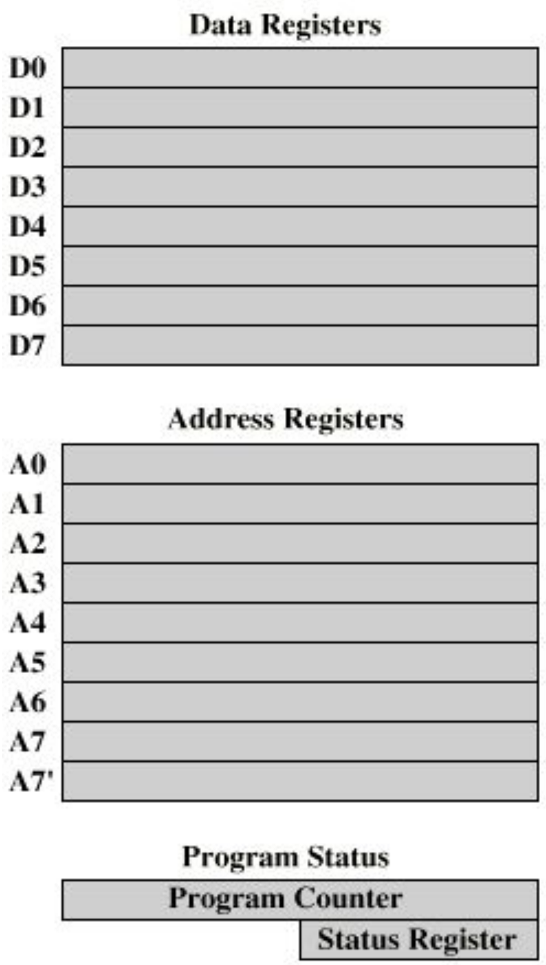
Control & Status Registers

- Program Counter (PC)
  - Contains the address of an instruction to be fetched
- Instruction Decoding Register (IR)
  - Contains the instruction most recently fetched
- Memory Address Register (MAR)
  - Contains the addres of location in memory
- Memory Buffer Register (MBR)
  - Contains a word or data to be written to memory or the word most recently read

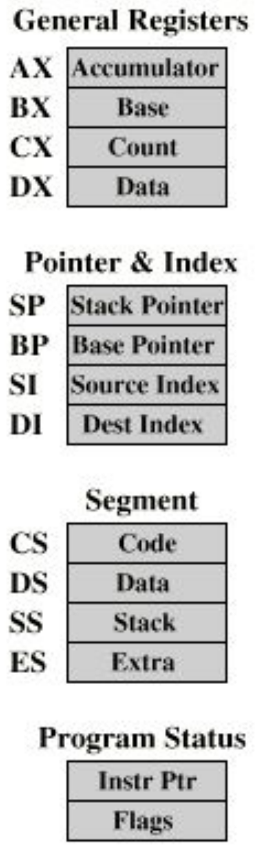
Program Status Word

- A set of bits containing status information
- Includes Condition Codes (flags)
  - Sign
    - sign of last result
  - Zero
    - set when the result is 0
  - Carry
    - set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high order bit
  - Equal
    - set if a logical compare result is equality
  - Overflow
    - used to indicate arithmetic overflow
  - Interrupt enable/disable
    - used to enable or disable interrupts

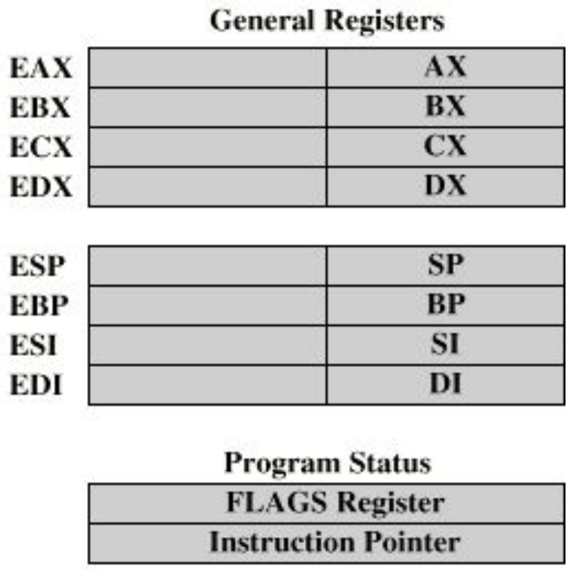
Example Register Organizations



(a) MC68000

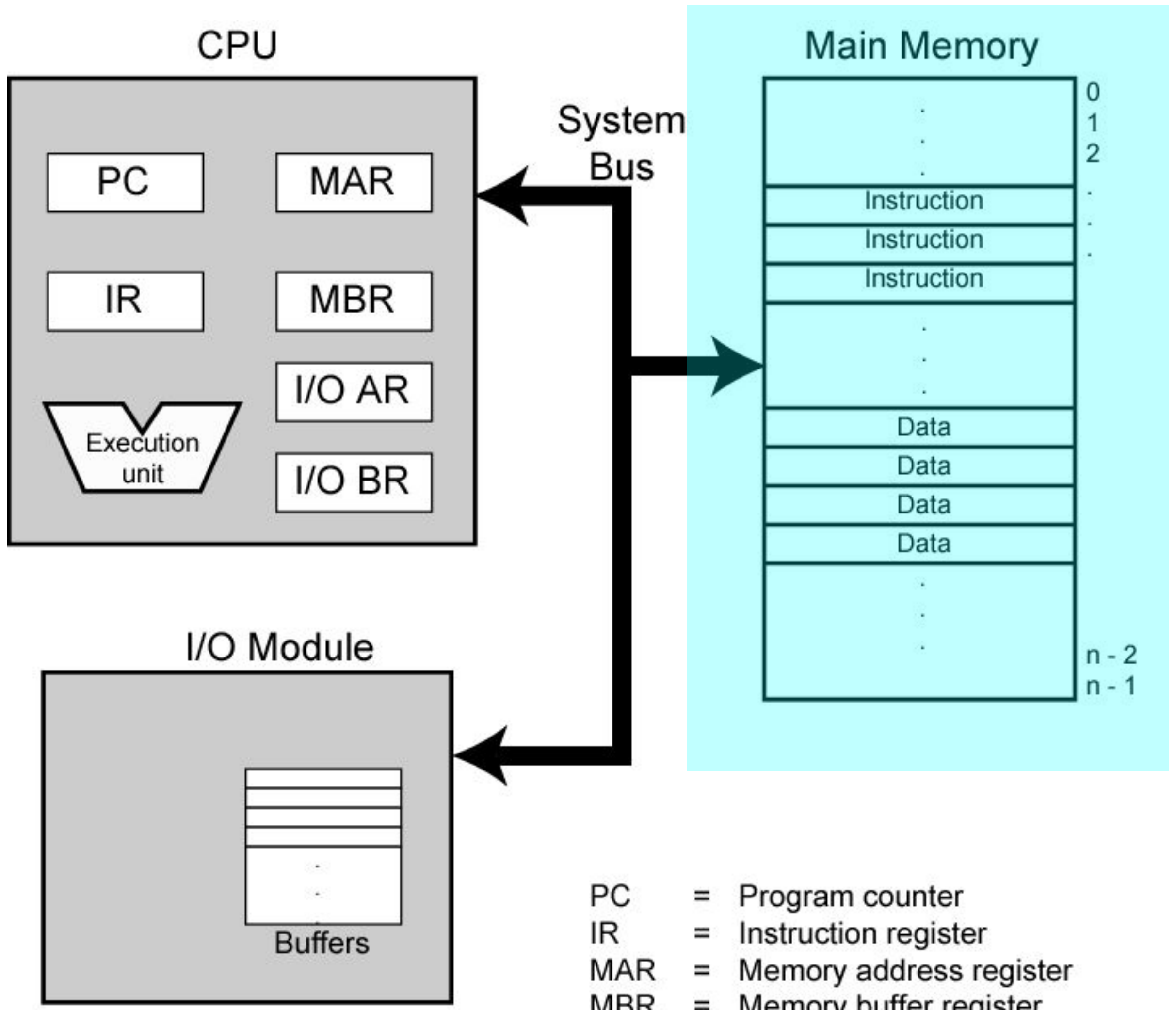


(b) 8086



(c) 80386 - Pentium II

Computer Components-Memory

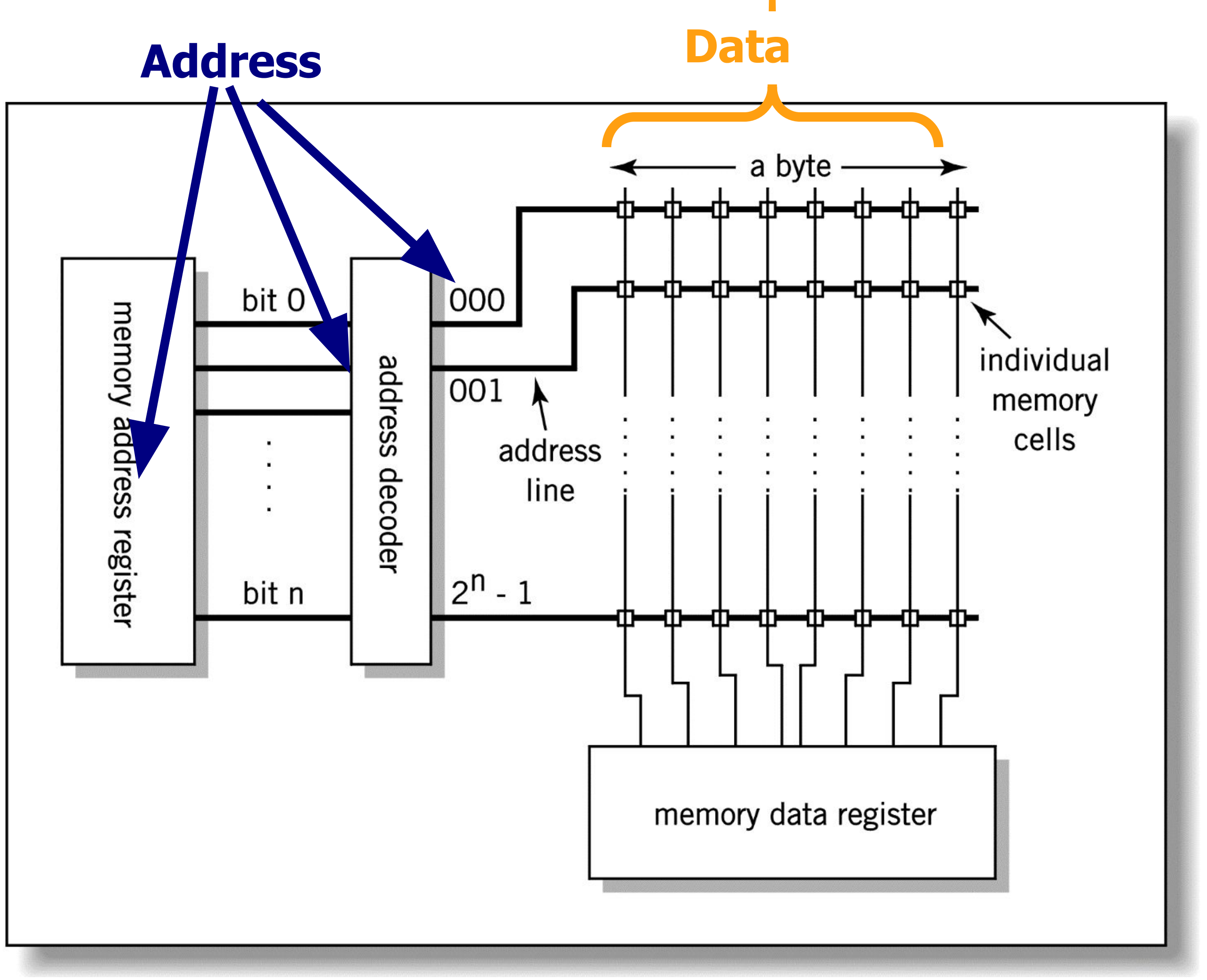


PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

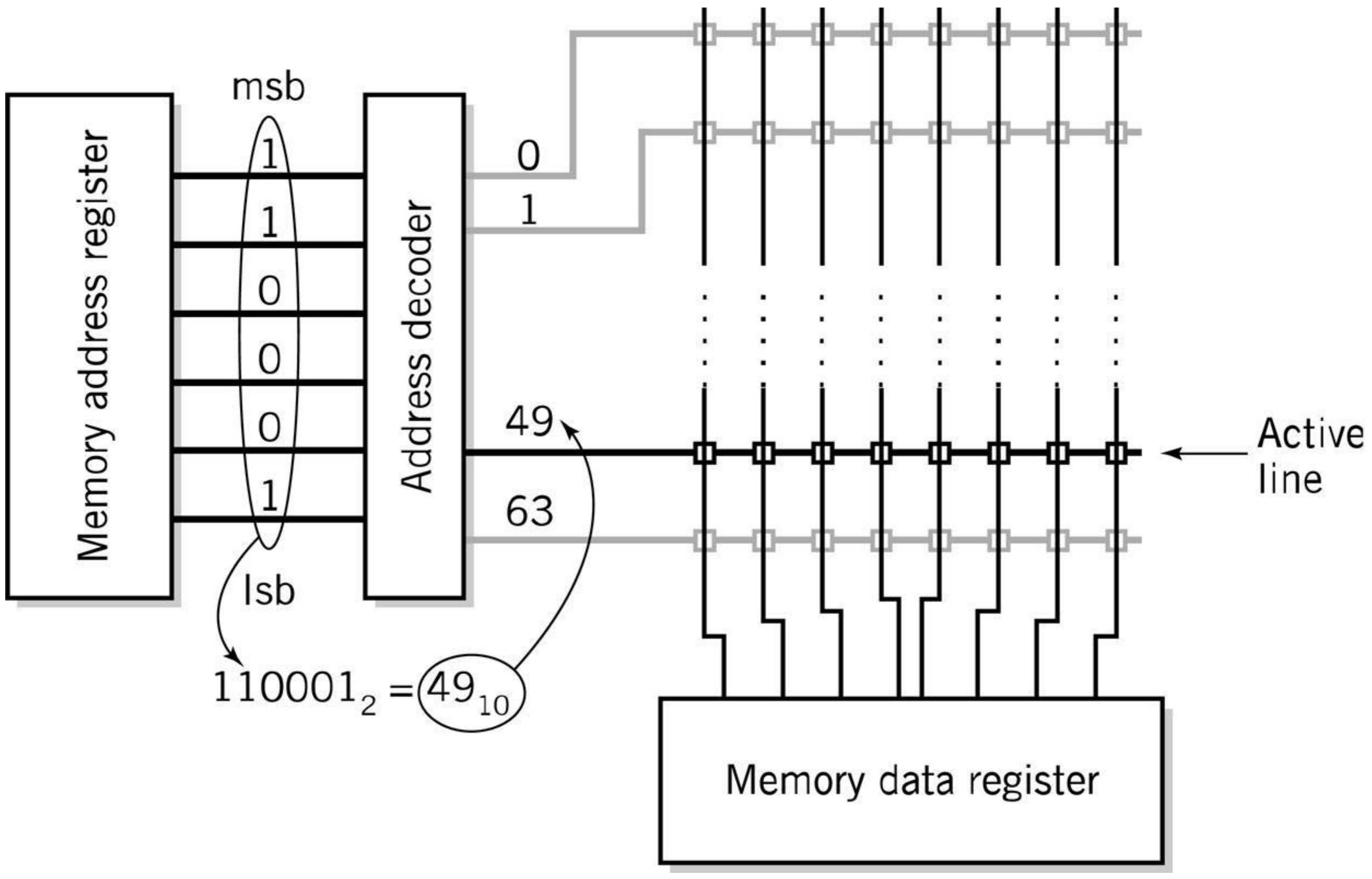
Operation of Memory

- Each memory location has a unique address
- Address from an instruction is copied to the MAR which finds the location in memory
- CPU determines if it is a store or retrieval
- Transfer takes place between the MBR and memory
  - MBR is a two way register

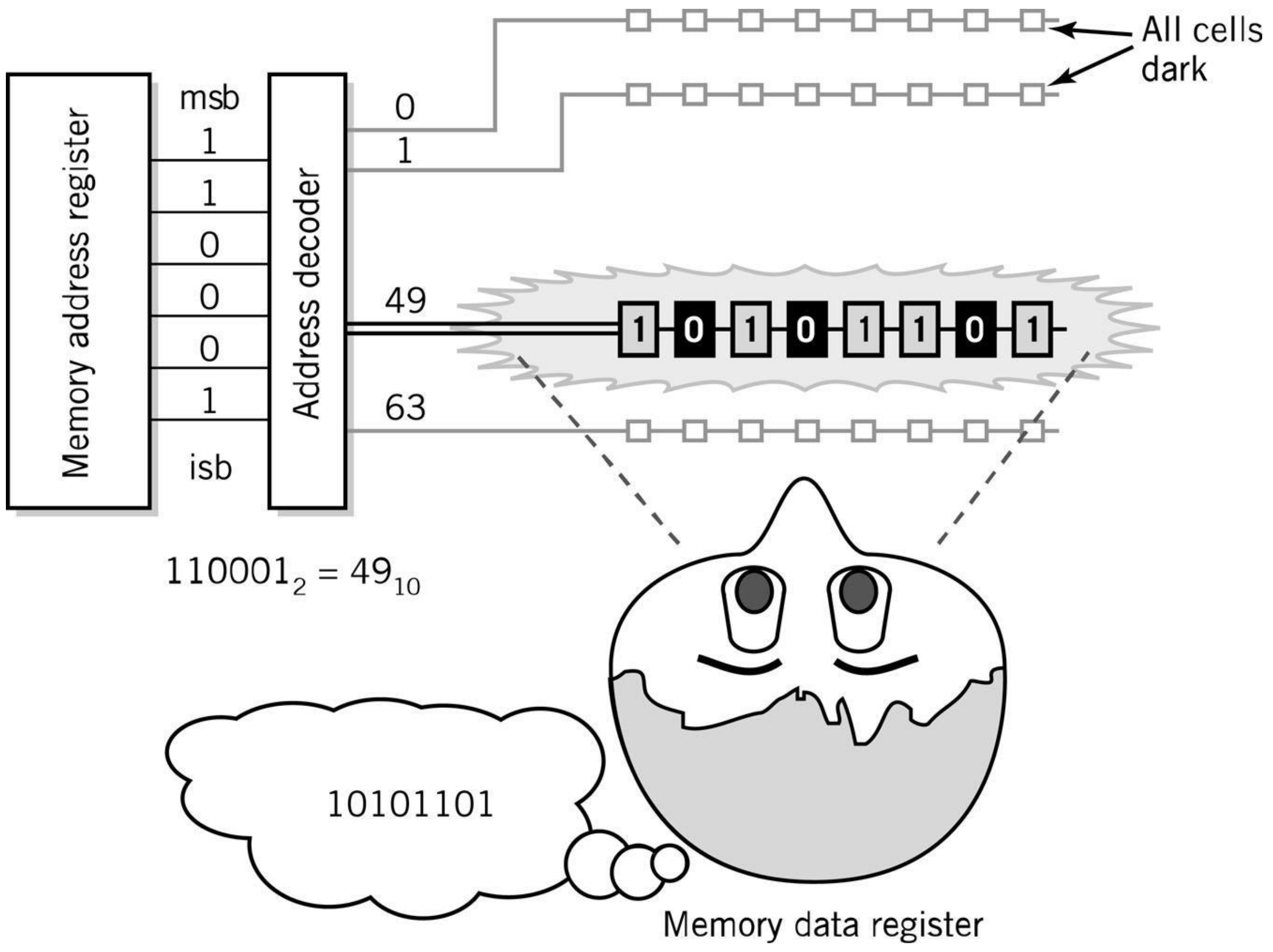
Relationship between MAR, MBR and Memory



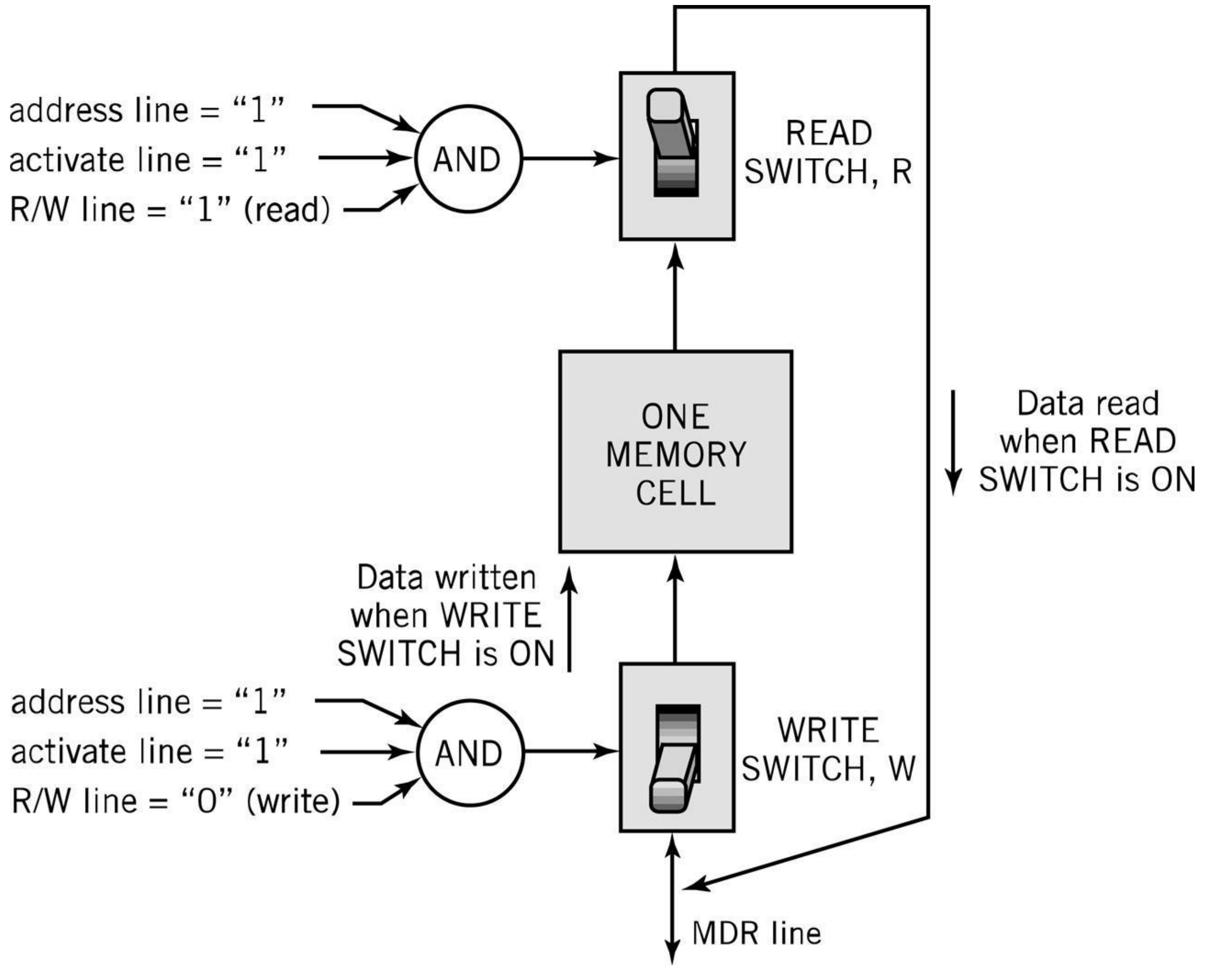
MAR-MBR Example



Visual Analogy of Memory



Individual Memory Cell



Memory Capacity

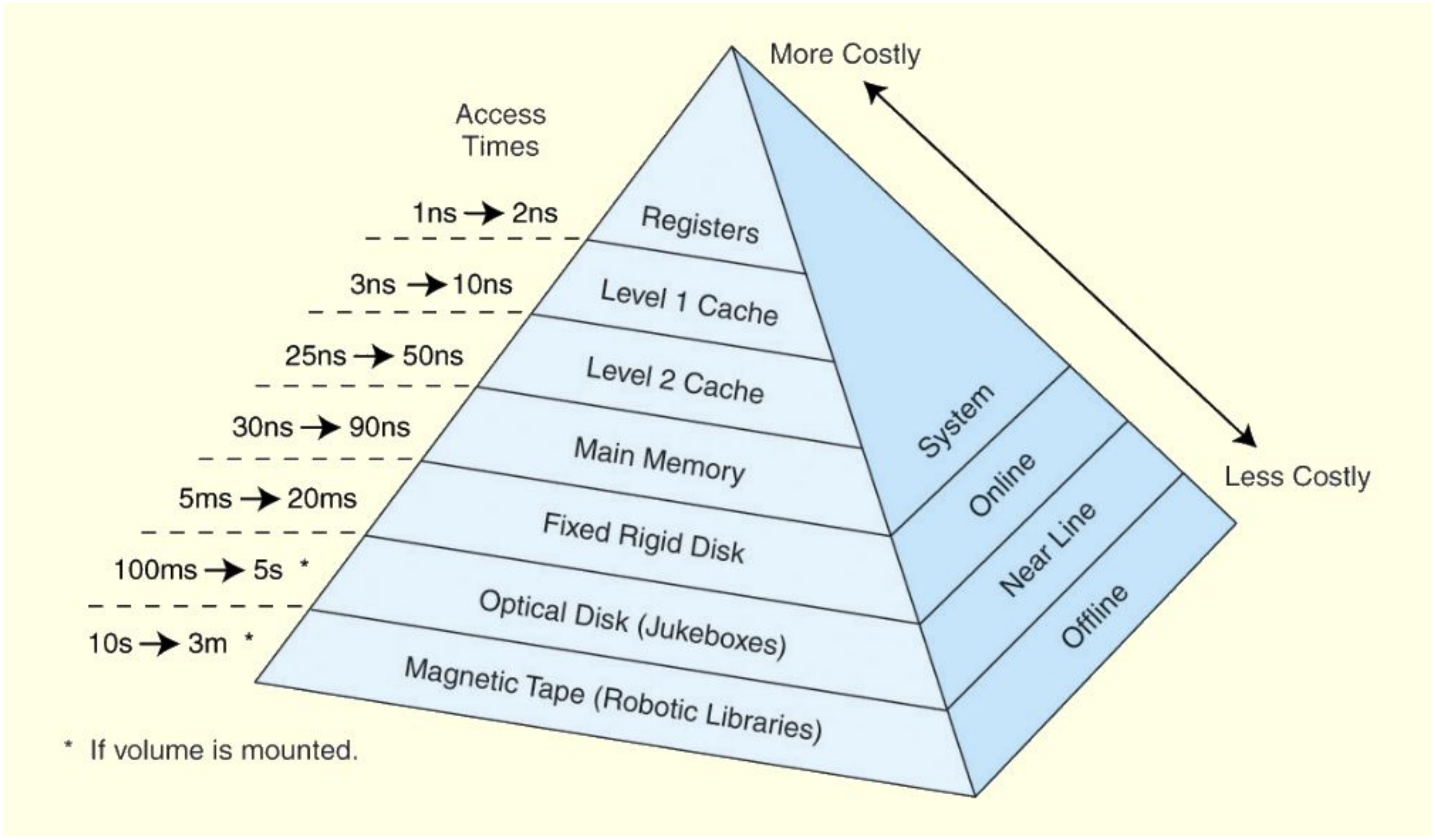
- Determined by two factors
  - Number of bits in the MAR
    - 2^K where K = width of the register in bits
  - Size of the address portion of the instruction
    - 4 bits allows 16 locations
    - 8 bits allows 256 locations
    - 32 bits allows 4,294,967,296 or 4 GB
- Important for performance
  - Insufficient memory can cause a processor to work at 50% below performance

Memory Hierarchy

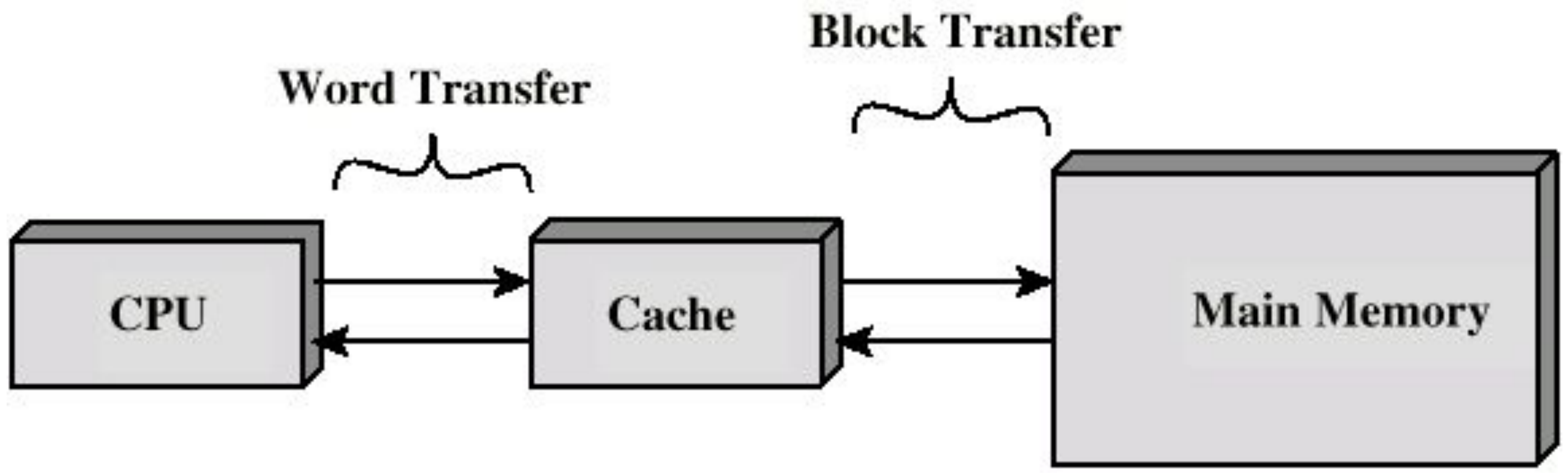
- Registers
  - In CPU
- Internal or Main memory
  - May include one or more levels of cache
  - “RAM”
- External memory
  - Backing store

Memory Hierarchy

- This storage organization can be thought of as a pyramid:



Cache



- Small amount of fast memory
- Sits between normal main memory and CPU
- May be located on CPU chip or module

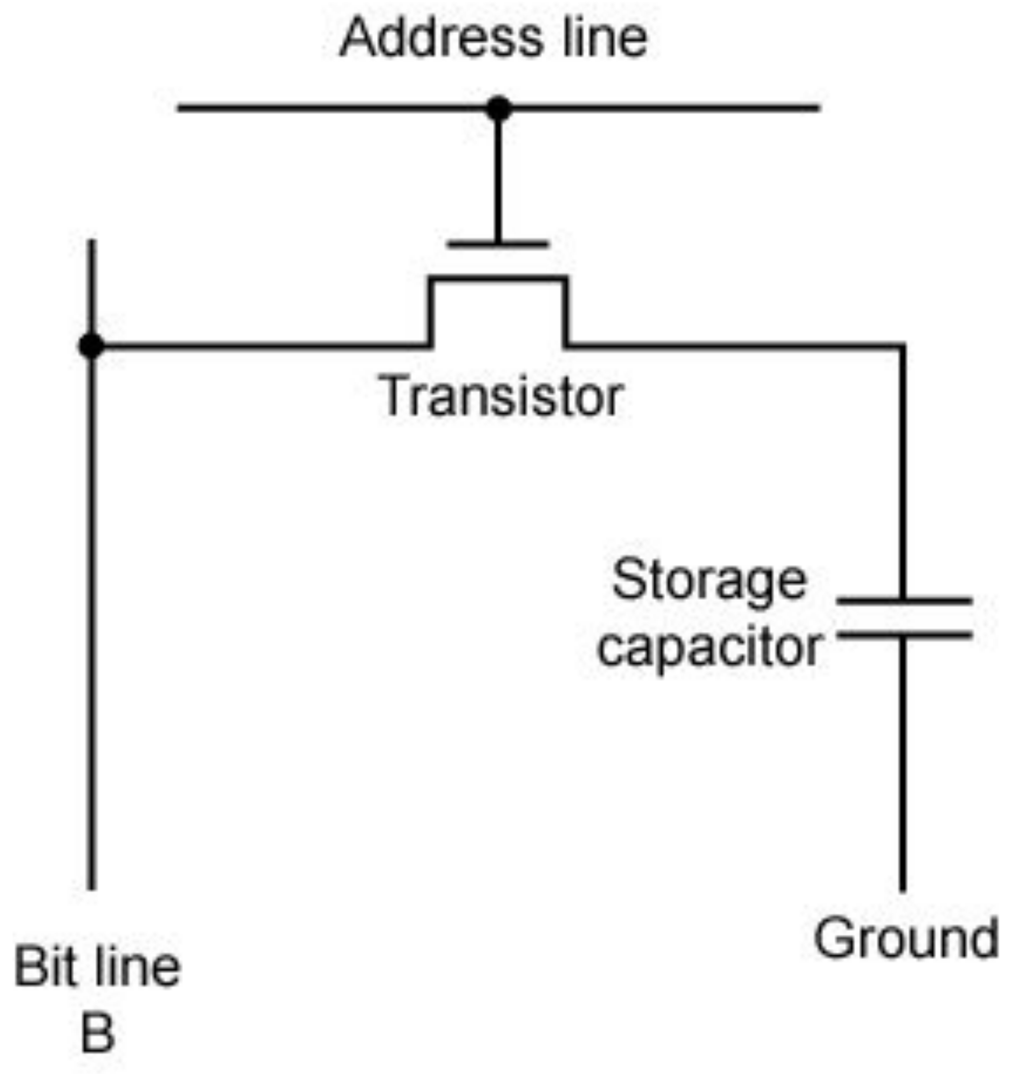
Virtual Memory

- Cache memory enhances performance by providing faster memory access speed.
- Virtual memory enhances performance by providing greater memory capacity, without the expense of adding main memory.
- Instead, a portion of a disk drive serves as an extension of main memory.
- If a system uses paging, virtual memory partitions main memory into individually managed *page frames*, that are written *(or paged)* to disk when they are not immediately needed.

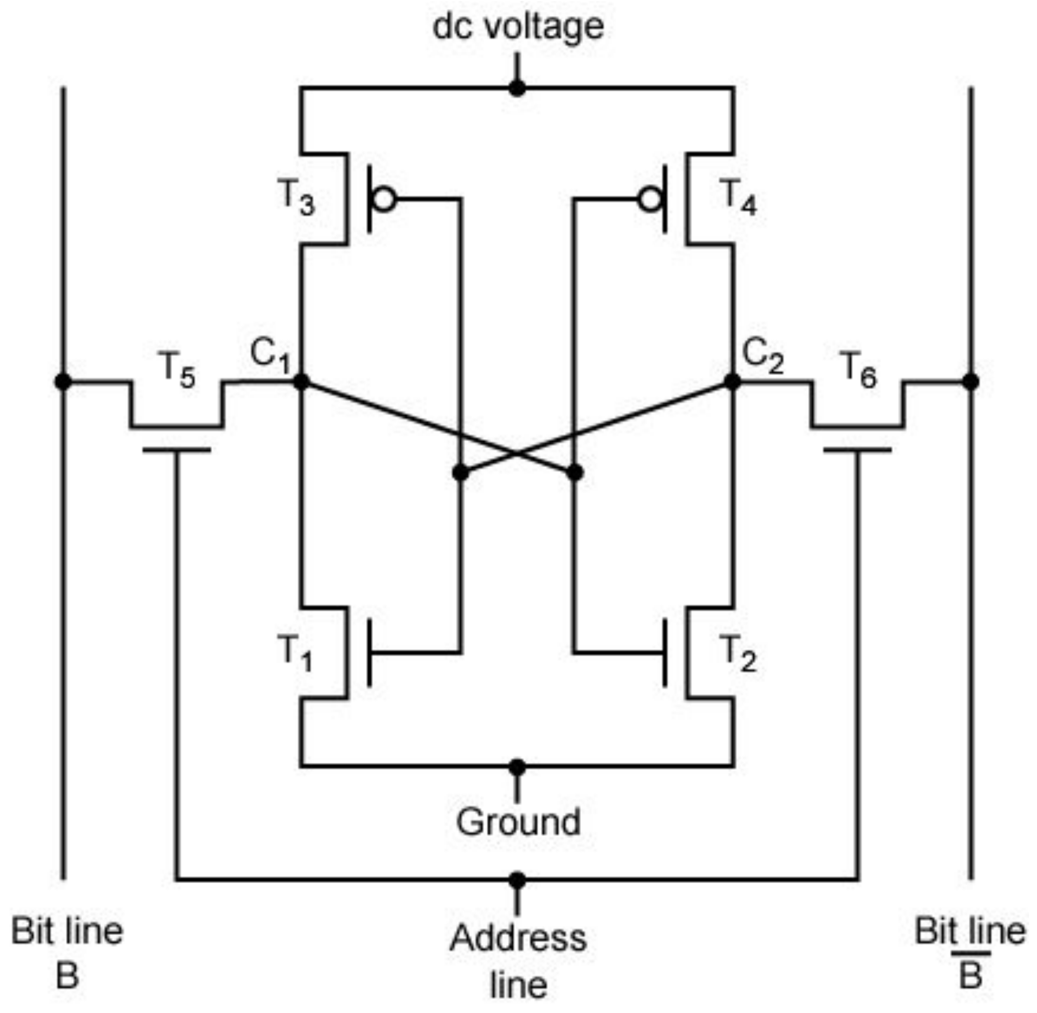
RAM: Random Access Memory

- DRAM (Dynamic RAM)
  - Most common, cheap
  - Volatile: must be refreshed (recharged with power) 1000’s of times each second
- SRAM (static RAM)
  - Faster than DRAM and more expensive than DRAM
  - Volatile
  - Frequently small amount used in *cache memory* for high-speed access used

Dynamic RAM Structure



Stating RAM Structure



Read Only Memory (ROM)

- Permanent storage
  - Nonvolatile
- Used in...
  - Microprogramming
  - Library subroutines
  - Systems programs (BIOS)
  - Function tables

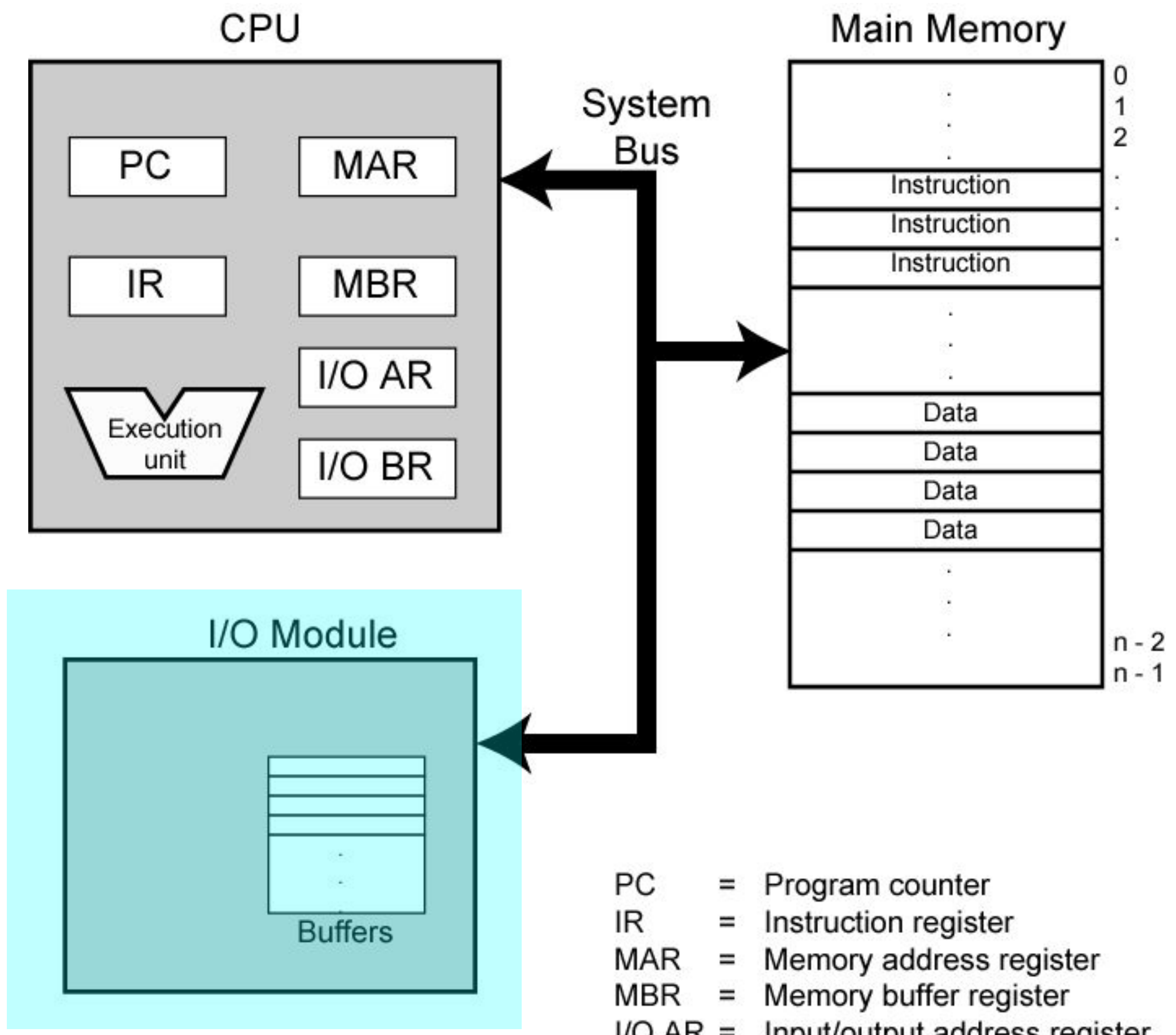
Types of ROM

- Written during manufacture
  - Very expensive for small runs
- Programmable (once)
  - PROM
  - Needs special equipment to program
- Read “mostly”
  - Erasable Programmable (EPROM)
    - Erased by UV
  - Electrically Erasable (EEPROM)
    - Takes much longer to write than read
  - Flash memory
    - Erase whole memory electrically

Types of External Memory

- SSD
  - Fast
  - Expensive (relatively)
- Magnetic Disk
  - RAID
  - Removable
- Optical
  - CD-ROM
  - CD-Recordable (CD-R)
  - CD-R/W
  - DVD
- Magnetic Tape

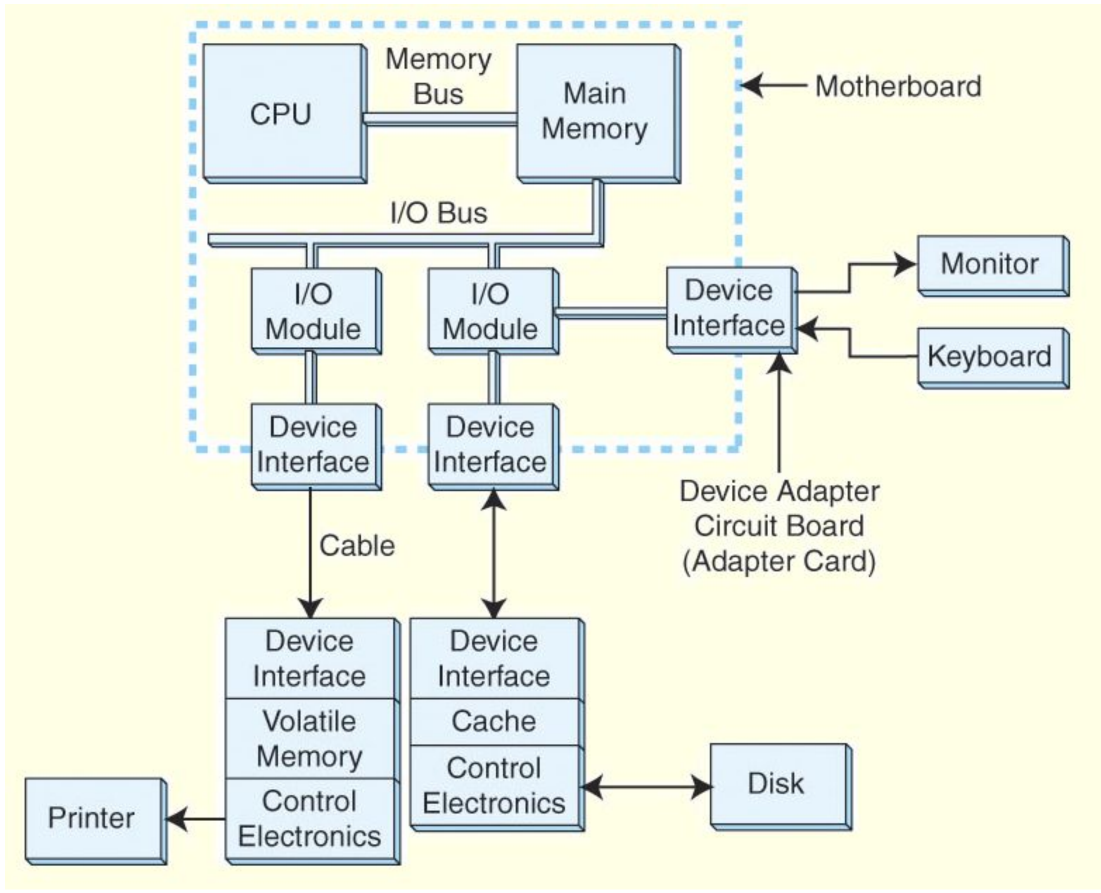
Computer Components-I/O



PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

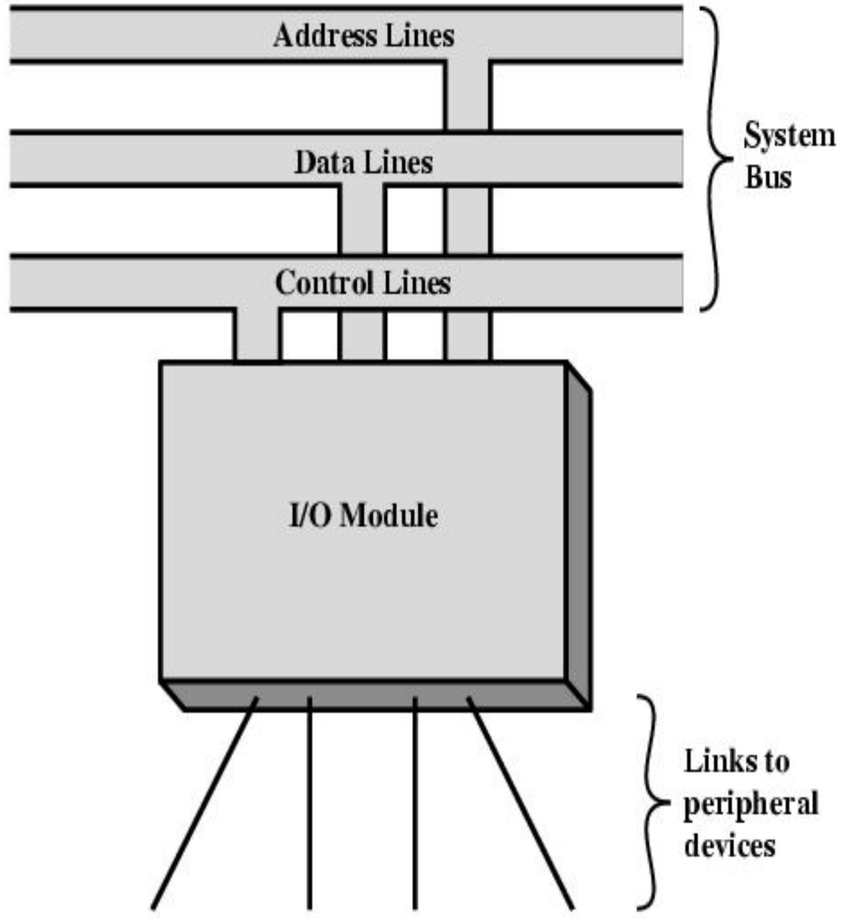
Input/Output Problems

- Wide variety of peripherals
  - Delivering different amounts of data
  - At different speeds
  - In different formats
- All slower than CPU and RAM
- Need I/O modules



Input/Output Module

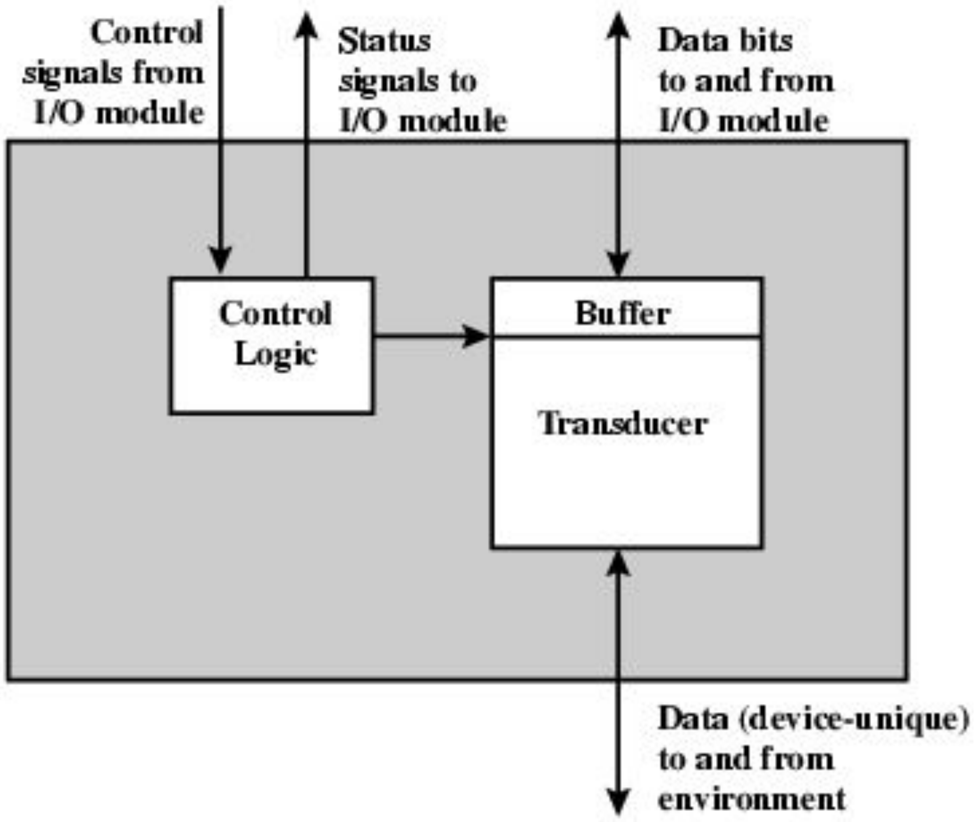
I/O Module Block Diagram



- Interface to CPU and Memory
- Interface to one or more peripherals
- I/O Module Function:
  - Control & Timing
  - CPU Communication
  - Device Communication
  - Data Buffering
  - Error Detection

External Devices

External Device Block Diagram


Control signals from I/O module
Status signals to I/O module
Data bits to and from I/O module
Control Logic
Buffer
Transducer
Data (device-unique) to and from environment

- External Devices:
  - Human readable
    - Screen, printer, keyboard
  - Machine readable
    - Monitoring and control
  - Communication
    - Modem
    - Network Interface Card (NIC)

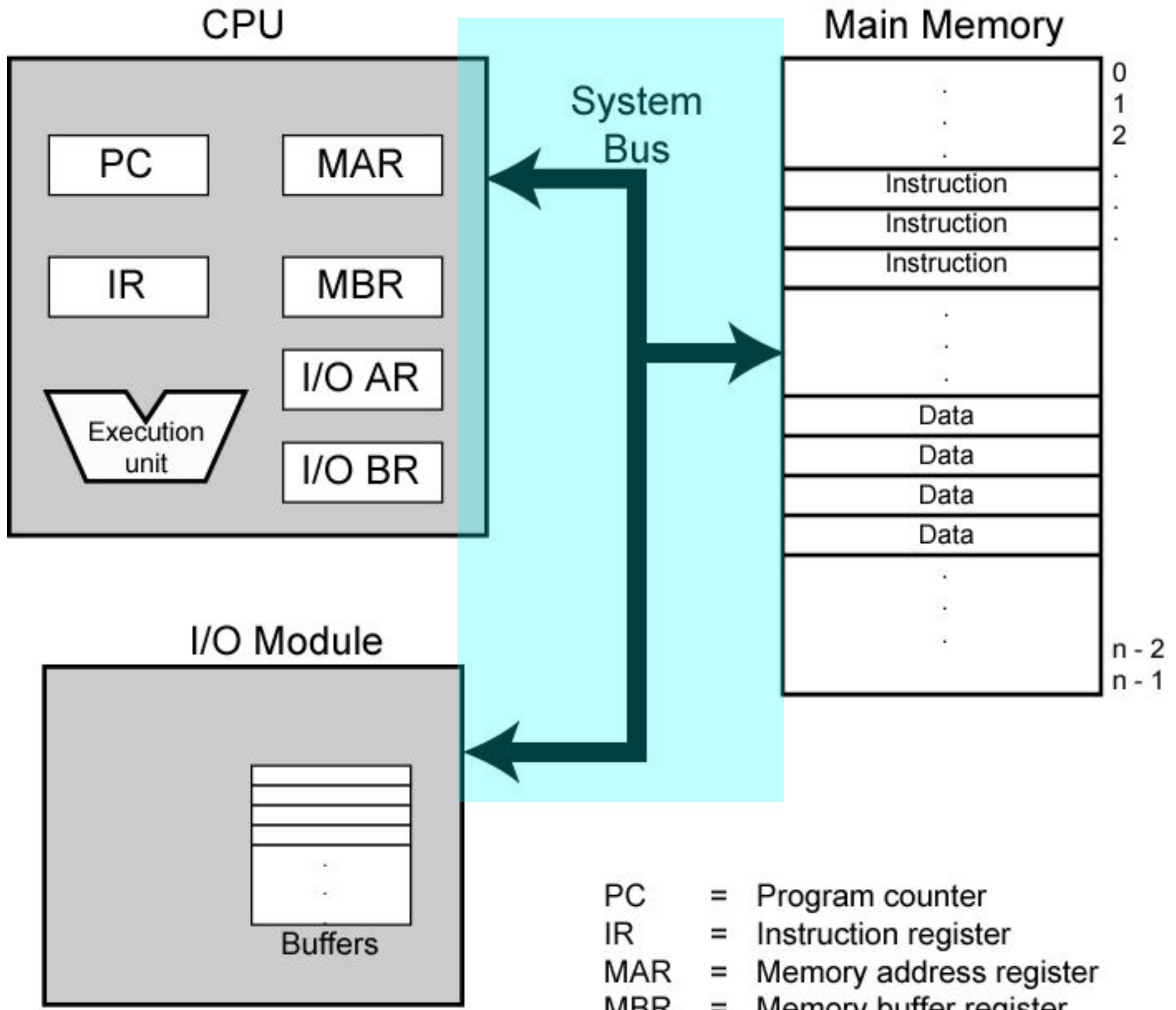
I/O Steps

- CPU checks I/O module device status
- I/O module returns status
- If ready, CPU requests data transfer
- I/O module gets data from device
- I/O module transfers data to CPU
- Variations for output, DMA, etc.

I/O Architectures

- I/O can be controlled in four general ways:
  - Programmed I/O
    - Reserves a register for each I/O device.
    - Each register is continually polled to detect data arrival.
  - Interrupt-Driven I/O
    - Allows the CPU to do other things until I/O is requested.
  - Direct Memory Access (DMA)
    - Offloads I/O processing to a special-purpose chip that takes care of the details.
  - Channel I/O
    - Uses dedicated I/O processors.

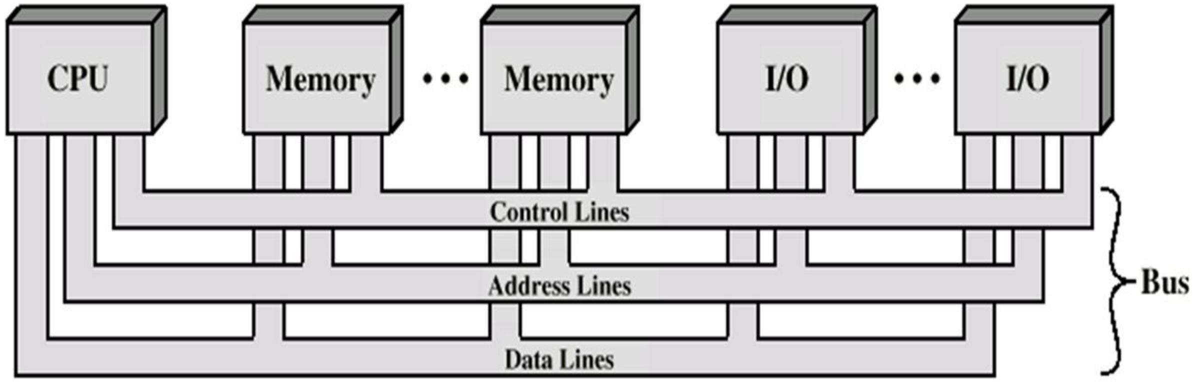
Computer Components- Bus



PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

Bus

- The physical connection that makes it possible to transfer data from one location in the computer system to another
- Group of electrical conductors for carrying signals from one location to another
- 4 kinds of signals
  - Data
    - Alphanumeric
    - Numerical
    - instructions
  - Addresses
  - Control signals
  - Power (sometimes)



Bus

- Connects CPU and Memory
- I/O peripherals: on the same bus as CPU/memory or separate bus
- Physical packaging commonly called *backplane*
  - Also called *system bus* or *external bus*
  - Example of *broadcast bus* (address bus)
  - Part of printed circuit board called *motherboard* that holds CPU and related components

Bus Characteristics

- Protocol
  - Documented agreement for communication
  - Specification that spells out the meaning of each line and each signal on each line
- Throughput, i.e., data transfer rate in bits per second
- Data width in bits carried simultaneously

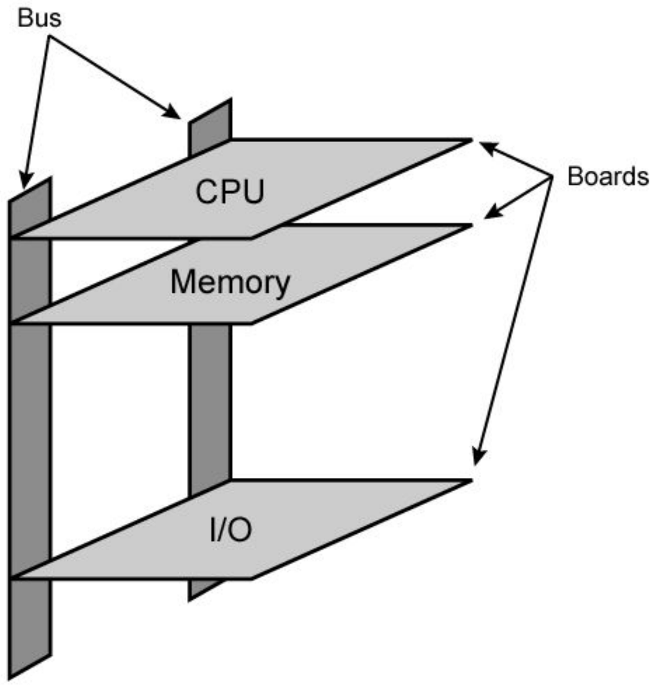
Bus types

- Data Bus
  - Carries data
  - Width is a key determinant of performance
    - 8, 16, 32, 64 bit
- Address bus
  - Identify the source or destination of data
  - Bus width determines maximum memory capacity of system
    - e.g. 8080 has 16 bit address bus giving 64k address space
- Control Bus
  - Control and timing information
    - Memory read/write; I/O read/write; Transfer acknowledge; Bus request; Bus grant; Interrupt request; Interrupt acknowledge; Clock; Reset

What do buses look like?

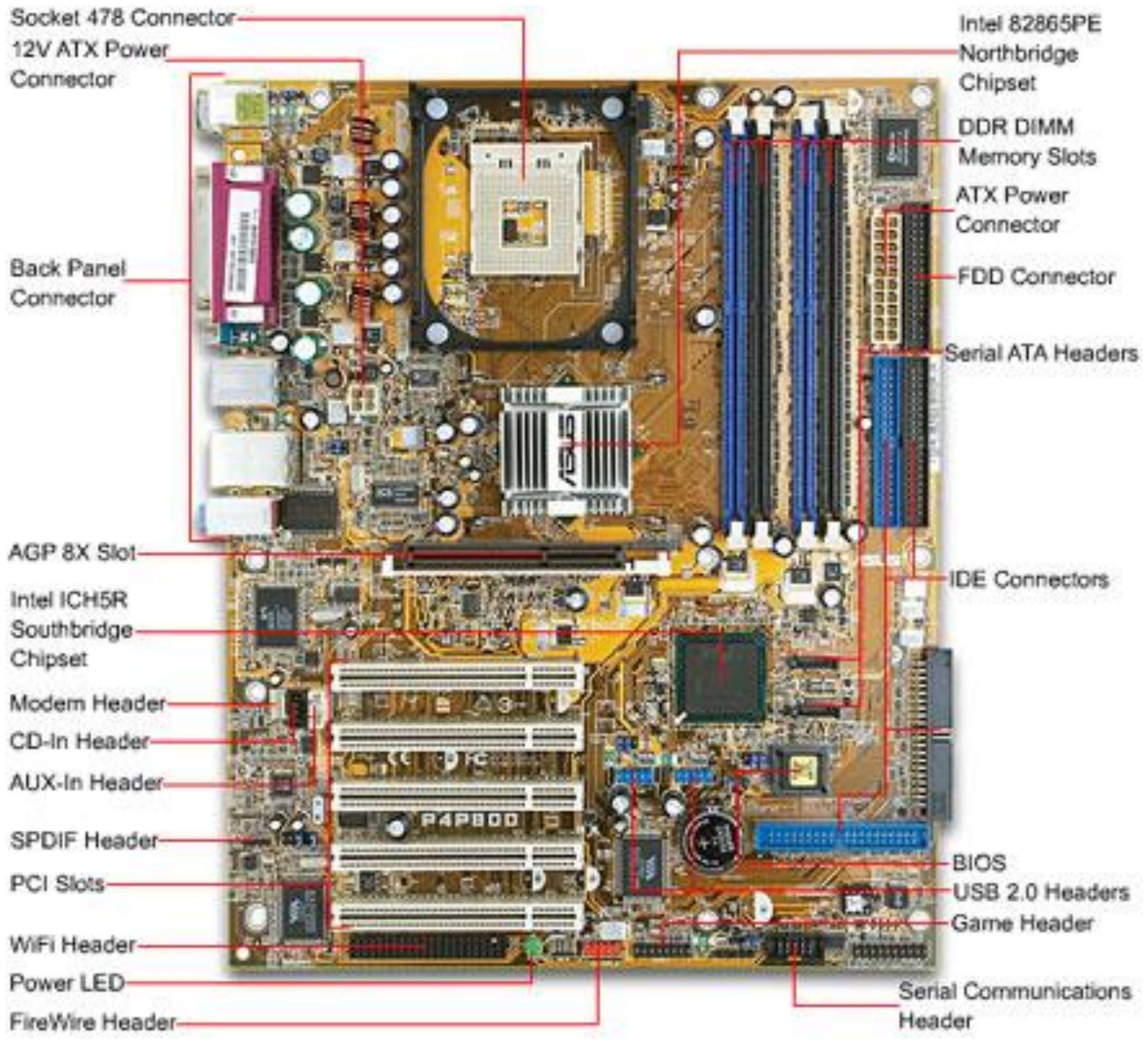
- Parallel lines on circuit boards
- Ribbon cables
- Strip connectors on mother boards
  - e.g. PCI
- Sets of wires

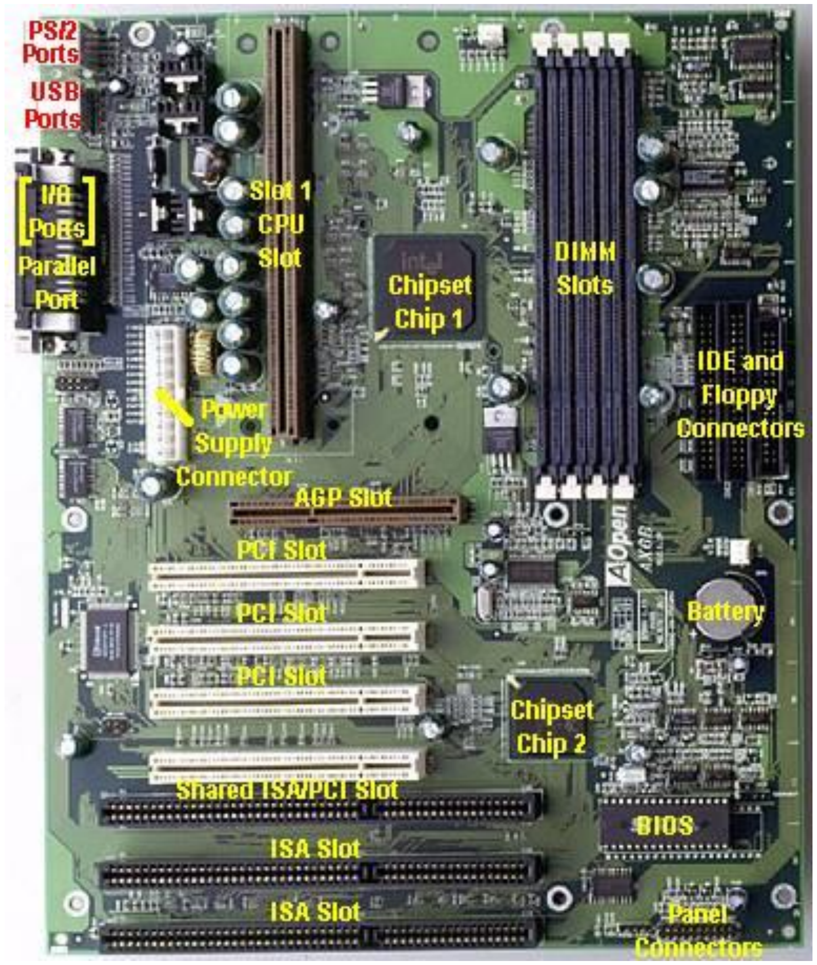
Physical Realization of Bus Architecture

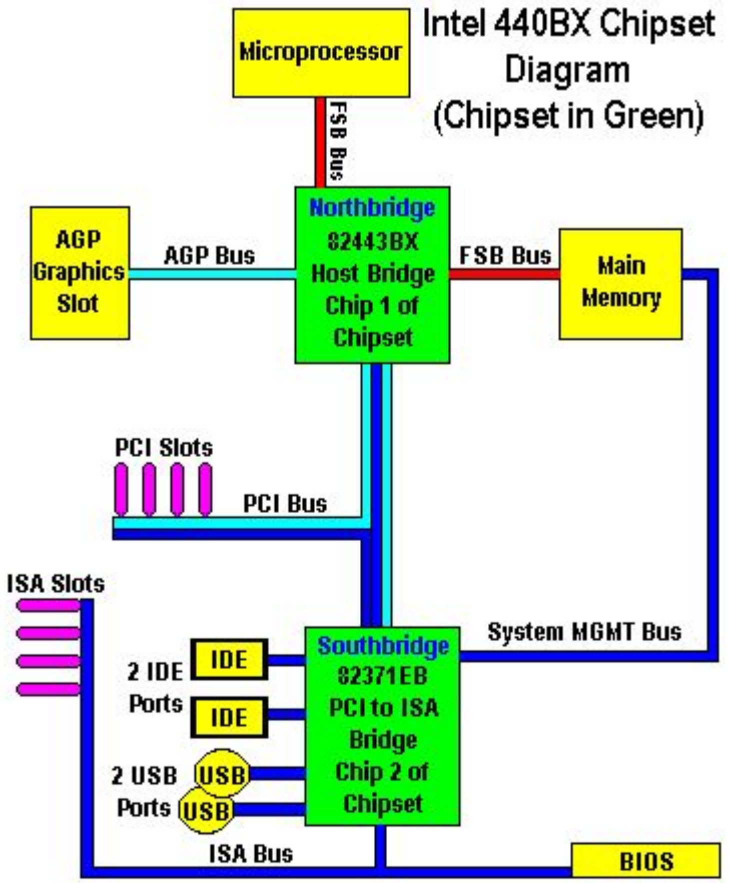


Motherboard

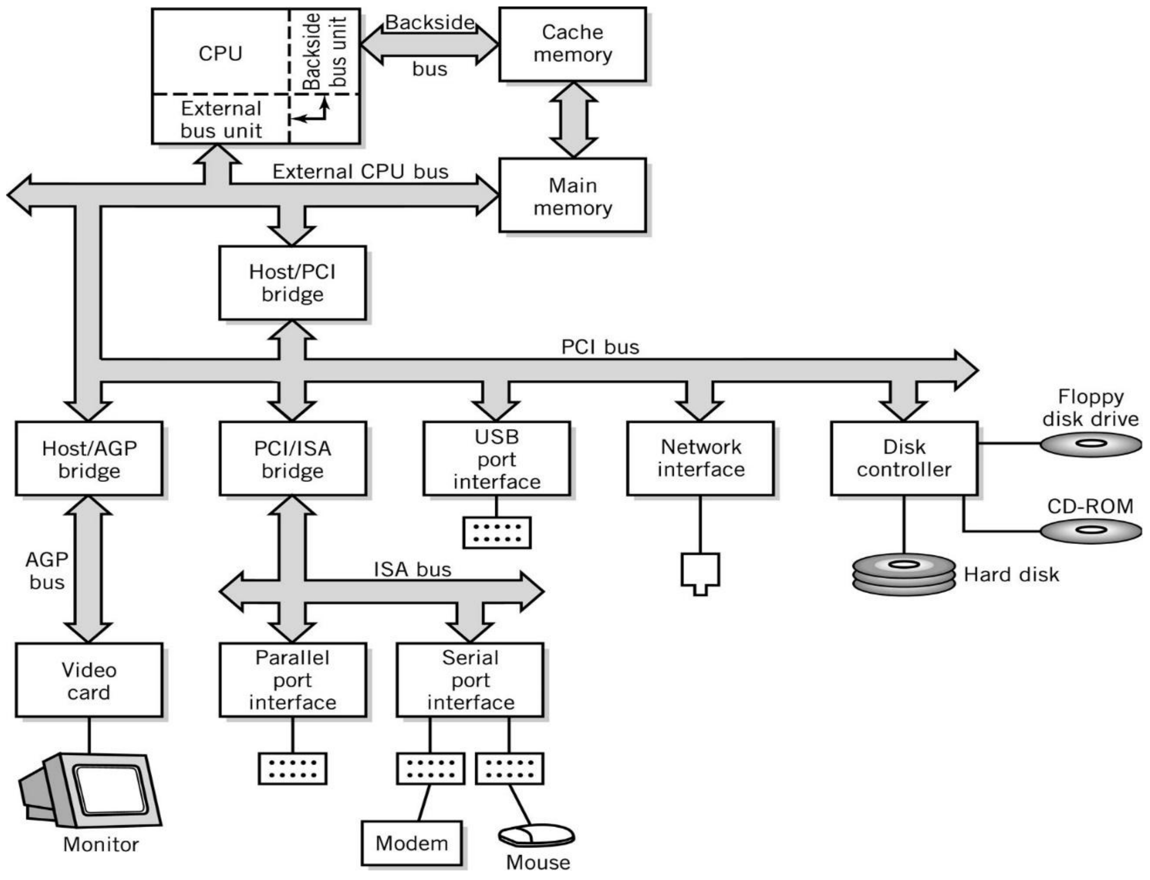
- Printed circuit board that holds CPU and related components including backplane




PS/2 Ports
USB Ports
I/O Ports
Parallel Port
Slot 1 CPU Slot
Chipset Chip 1
DIMM Slots
IDE and Floppy Connectors
Power Supply Connector
AGP Slot
PCI Slot
PCI Slot
PCI Slot
Battery
Chipset Chip 2
Shared ISA/PCI Slot
BIOS
ISA Slot
ISA Slot
Panel Connectors


Intel 440BX Chipset Diagram (Chipset in Green)
Microprocessor
FSB Bus
AGP Graphics Slot
AGP Bus
Northbridge 82443BX Host Bridge Chip 1 of Chipset
FSB Bus
Main Memory
PCI Slots
PCI Bus
ISA Slots
System MGMT Bus
2 IDE Ports
IDE
IDE
2 USB Ports
USB
USB
Southbridge 82371EB PCI to ISA Bridge Chip 2 of Chipset
ISA Bus
BIOS

Typical PC Interconnections



Bus interface bridges connect different bus types

Instructions

- Instruction:
  - Direction given to a computer
  - Causes electrical signals to be sent through specific circuits for processing
- The operation of the processor is determined by the instructions it executes,
  - which is referrred as machine instructions or computer instructions
- The collection of different instructions that the processor execute is referred as the processor's instruction set.

What is an Instruction Set?

- The complete collection of instructions that are understood by a CPU
  - Machine Code
  - Binary
- Usually represented by assembly codes
- Differentiates computer architecture by the
  - Number of instructions
  - Complexity of operations performed by individual instructions
  - Data types supported
  - Format (layout, fixed vs. variable length)
  - Use of registers
  - Addressing (size, modes)

Instruction Elements

OPCODE	Source OPERAND	Result OPERAND

- Operation code (OPCODE): task
  - Do this
- Source OPERAND reference(s)
  - To this
- Result OPERAND reference
  - Put the answer here
- Next Instruction Reference
  - When you have done that, do this...

$\left.\begin{array}{l}\text{Source OPERAND reference(s)}\\ \text{Result OPERAND reference}\end{array}\right\}$ Addresses

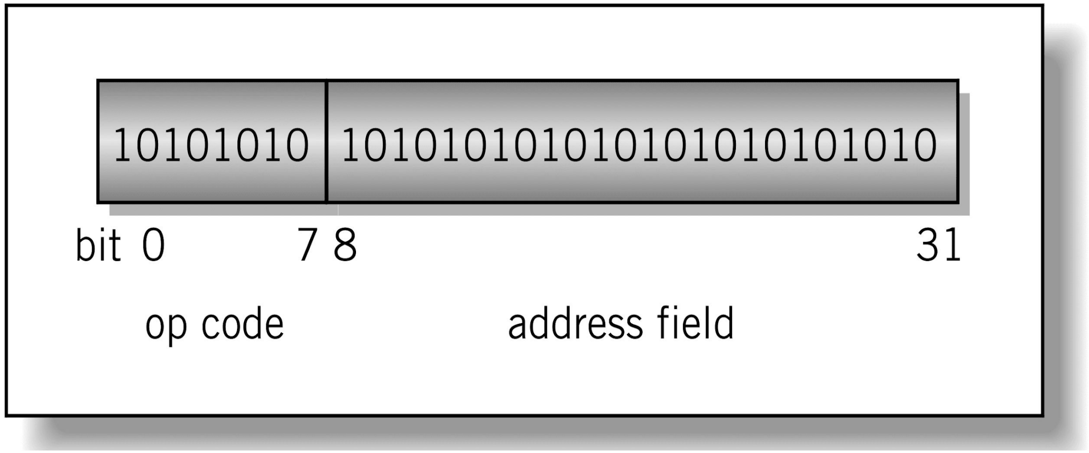
Instruction Elements

- Source and Result Operands can be in one of the following areas:
  - Main memory
  - Virtual memory
  - Cache
  - CPU register
  - I/O device

Instruction Format

- *Machine-specific* template that specifies
  - Length of the op code
  - Number of operands
  - Length of operands

Simple 32-bit Instruction Format



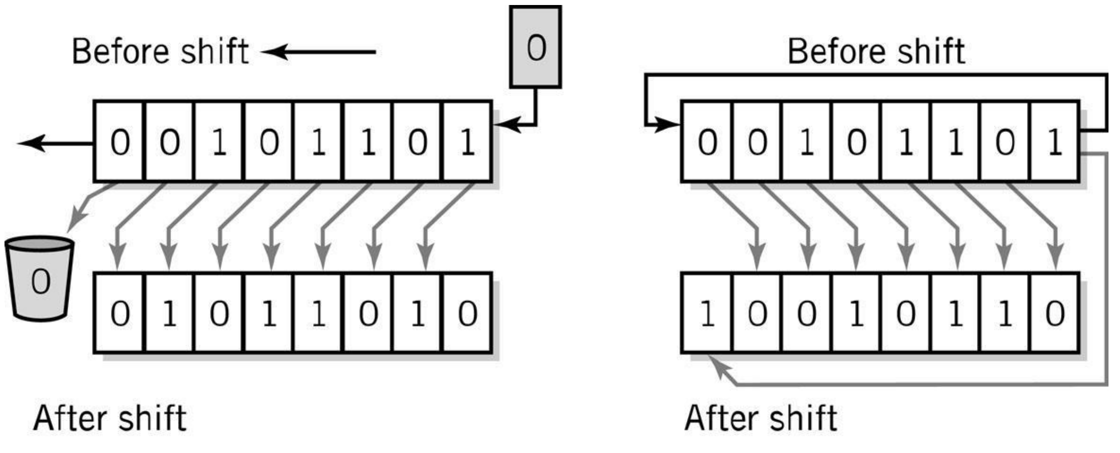
Instruction Types

- Data Transfer (load, store)
  - Most common, greatest flexibility
  - Involve memory and registers
  - What's a *word* ? 16? 32? 64 bits?
- Arithmetic
  - Operators `+ - / * ^`
  - Integers and floating point
- Logical or Boolean
  - Relational operators: `> < =`
  - Boolean operators **AND, OR, XOR, NOR,** and **NOT**
- Single operand manipulation instructions
  - Negating, decrementing, incrementing

More Instruction Types

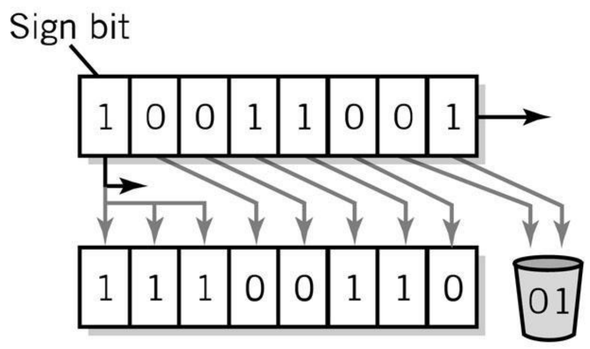
- Bit manipulation instructions
  - Flags to test for conditions
- Shift and rotate
- Program control
- Stack instructions
- Multiple data instructions
- I/O and machine control

Register Shifts and Rotates



a. Left logical shift register 1 bit

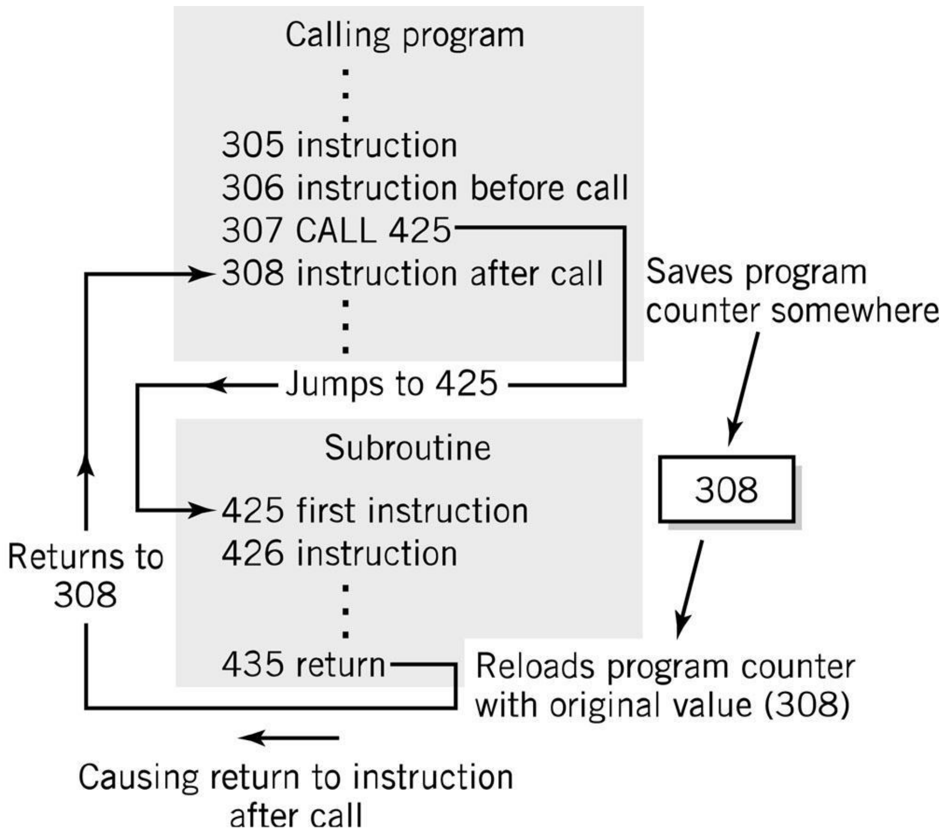
b. Rotate right 1 bit



c. Right arithmetic shift 2 bits

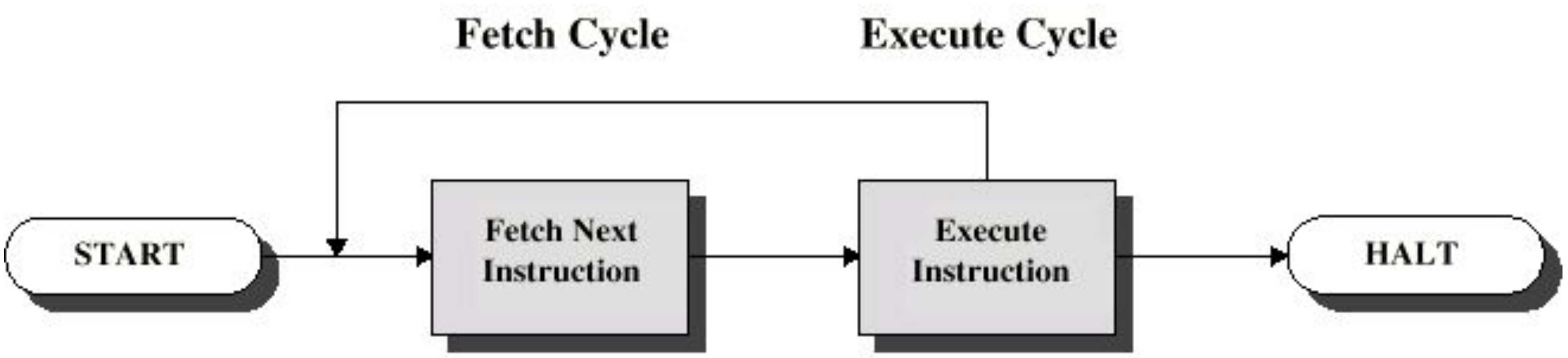
Program Control Instructions

- Program control
  - Jump and branch
  - Subroutine call and return


Calling program
305 instruction
306 instruction before call
307 CALL 425
308 instruction after call
Saves program counter somewhere
Jumps to 425
Subroutine
308
425 first instruction
426 instruction
Returns to 308
435 return
Reloads program counter with original value (308)
Causing return to instruction after call

Instruction Cycle

- Two steps:
  - Fetch
  - Execute



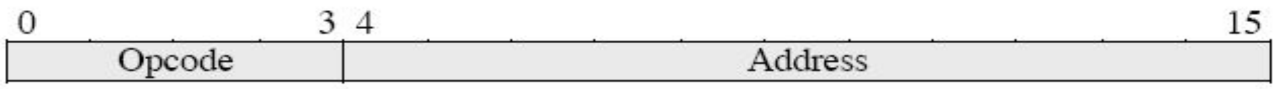
Fetch Cycle

- Program Counter (PC) holds address of next instruction to fetch
- Processor fetches instruction from memory location pointed to by PC
- Increment PC
  - Unless told otherwise
- Instruction loaded into Instruction Register (IR)
- Processor interprets instruction and performs required actions

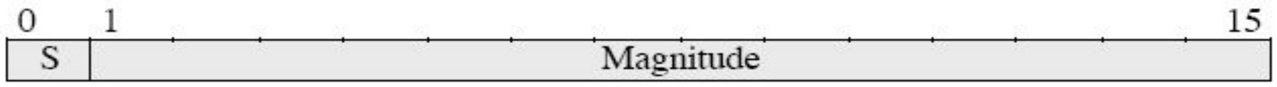
Execute Cycle

- Processor-memory
  - data transfer between CPU and main memory
- Processor I/O
  - Data transfer between CPU and I/O module
- Data processing
  - Some arithmetic or logical operation on data
- Control
  - Alteration of sequence of operations
  - e.g. jump
- Combination of above

A simple example – A hypotetical machine



(a) Instruction format



(b) Integer format

Program Counter (PC) = Address of instruction
Instruction Register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

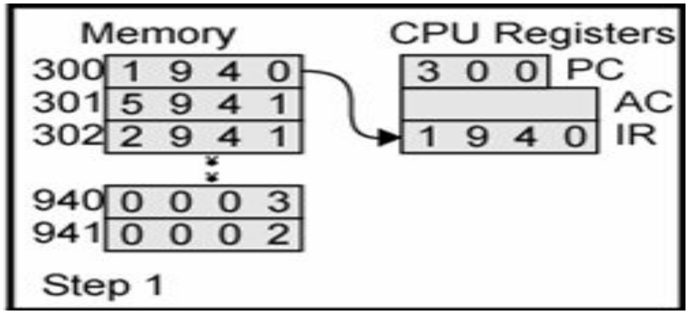
0001 = Load AC from Memory
0010 = Store AC to Memory
0101 = Add to AC from Memory

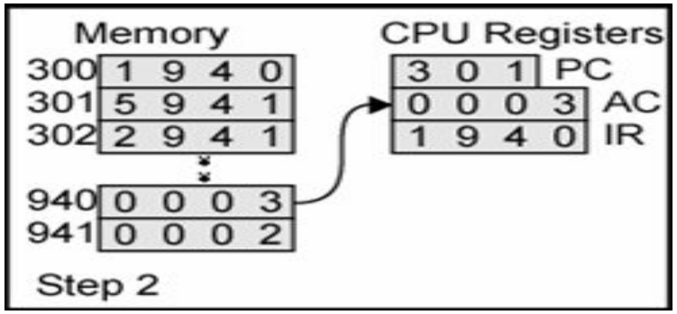
(d) Partial list of opcodes

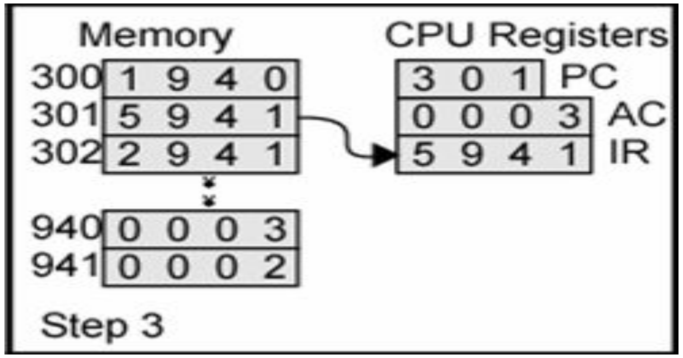
A simple example –

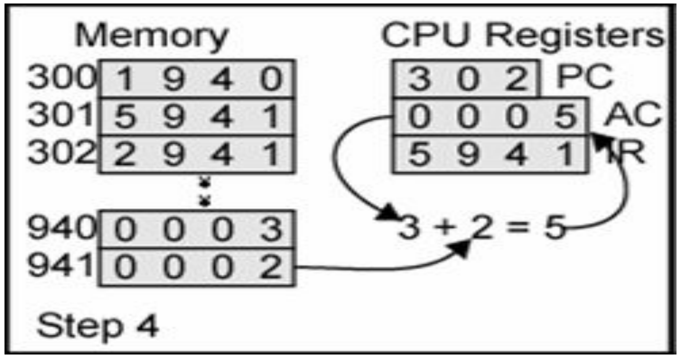
- Next figure illustrates a partial program execution.
- It adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the address 941.
- Here 3 instructions (3 fetch and 3 execute cycles) are required

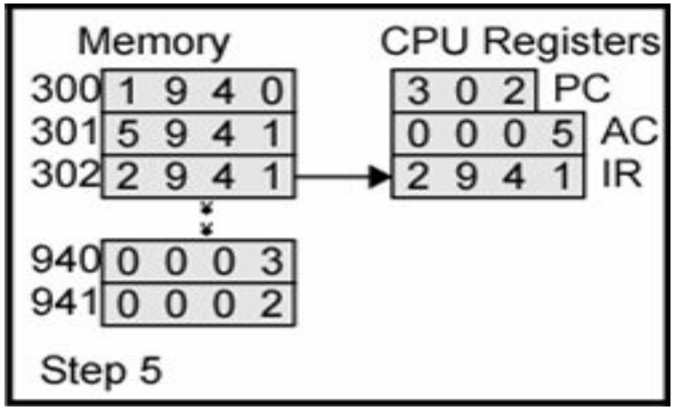
Example of Program Execution

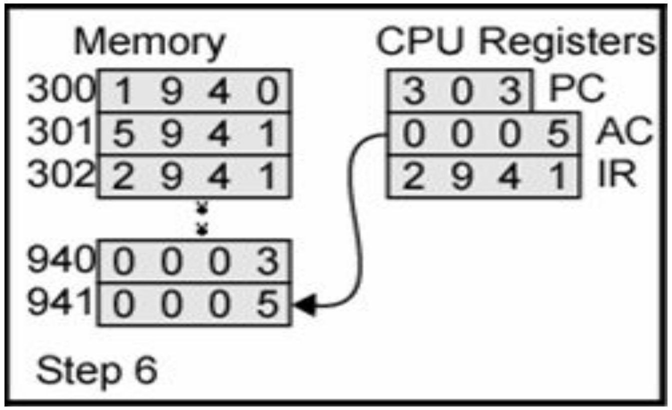










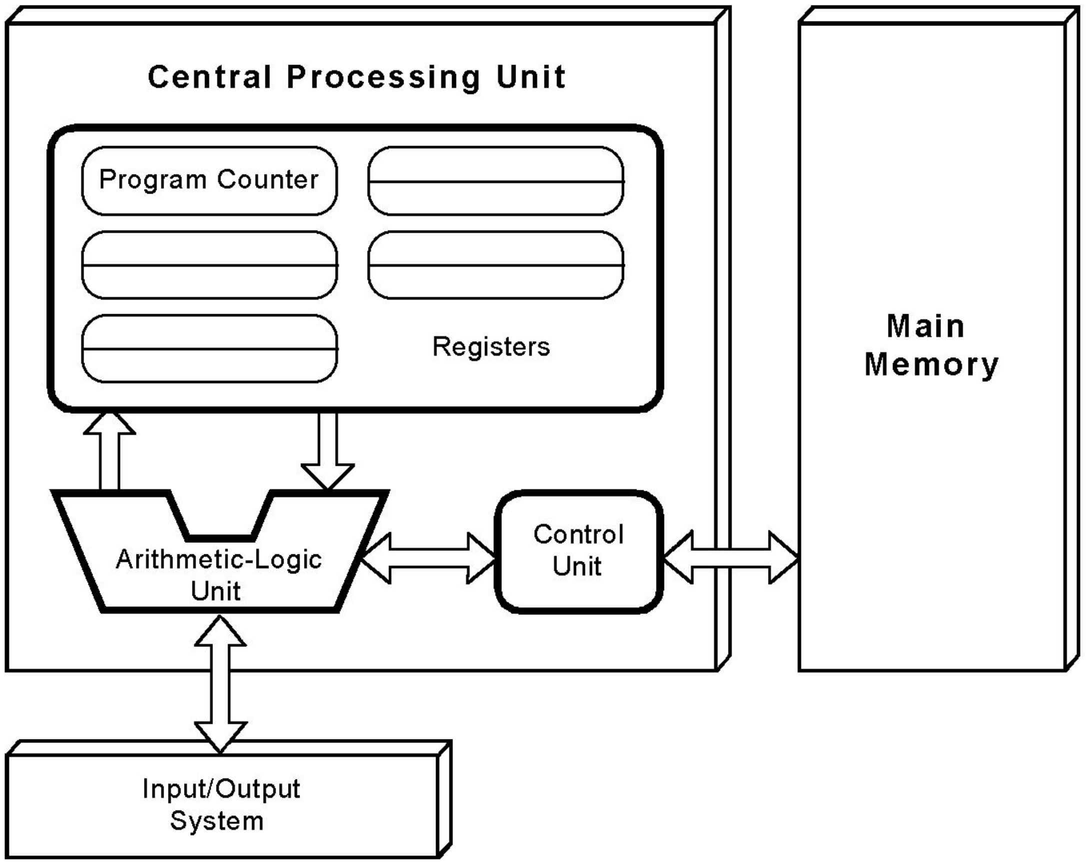


A virtual processor for understanding instruction cycle

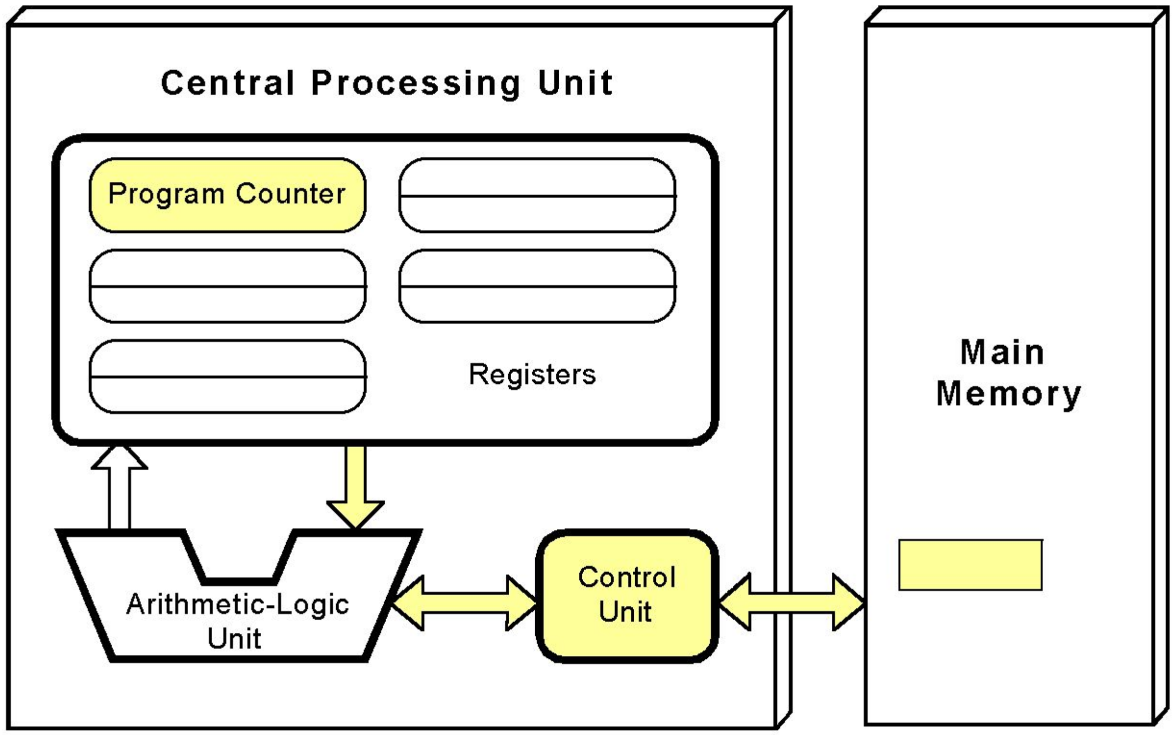
The Visible Virtual Machine (VVM)

The von Neumann Model

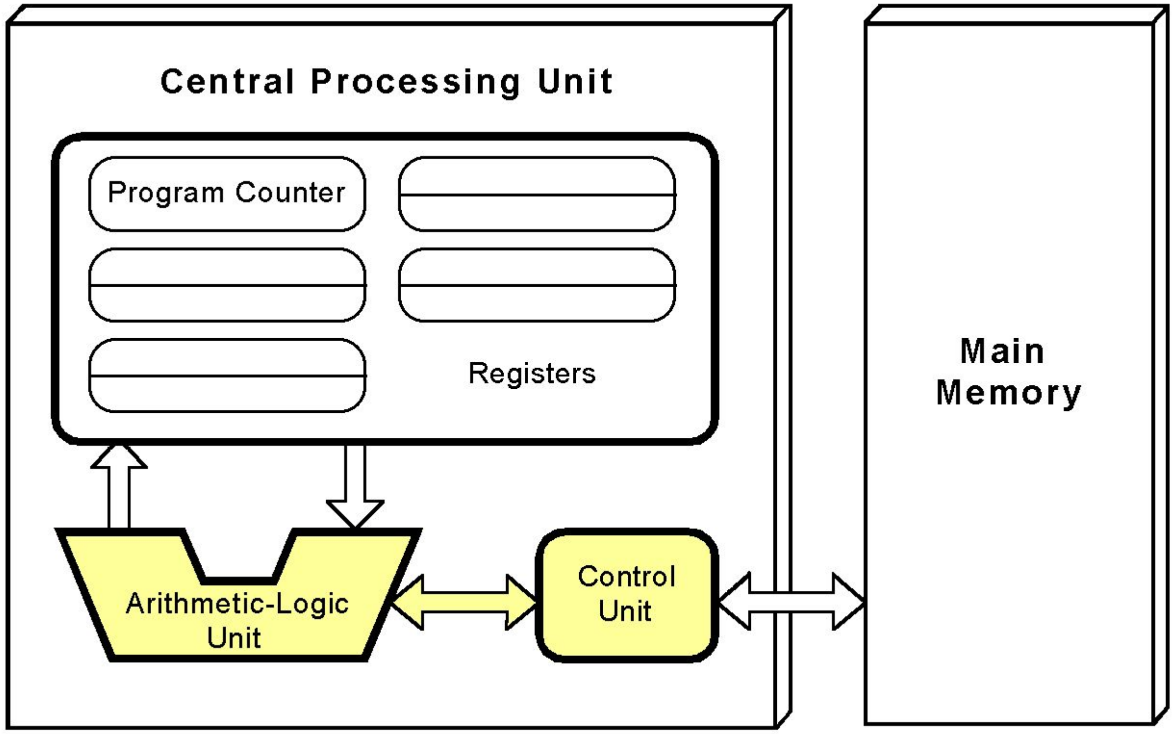
- This is a general depiction of a von Neumann system:
- These computers employ a fetch-decode-execute cycle to run programs as follows . . .



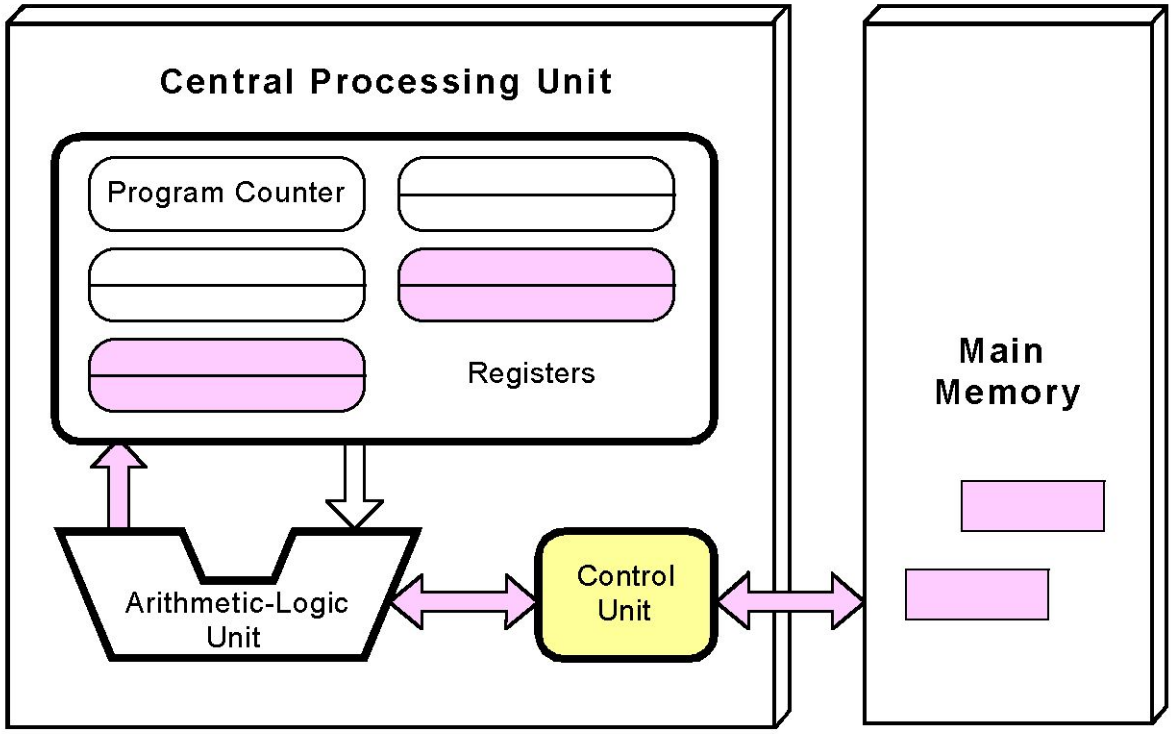
- **The control unit fetches the next instruction from memory using the program counter to determine where the instruction is located.**



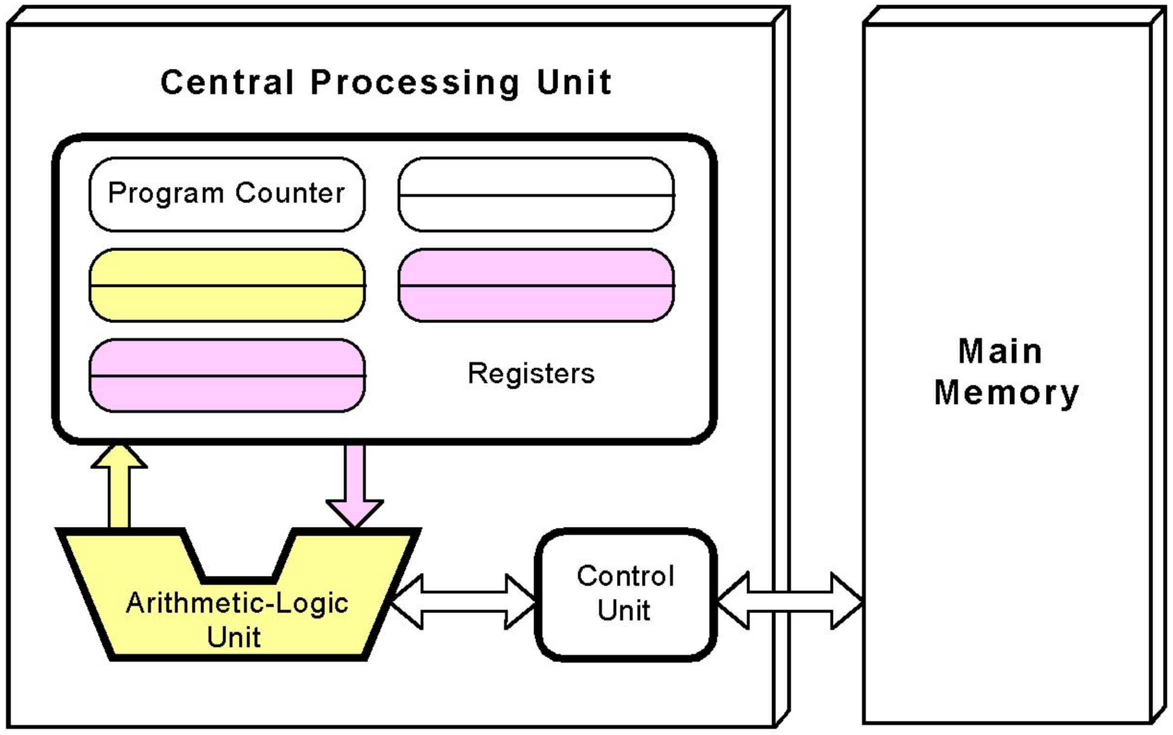
- **The instruction is decoded into a language that the ALU can understand.**



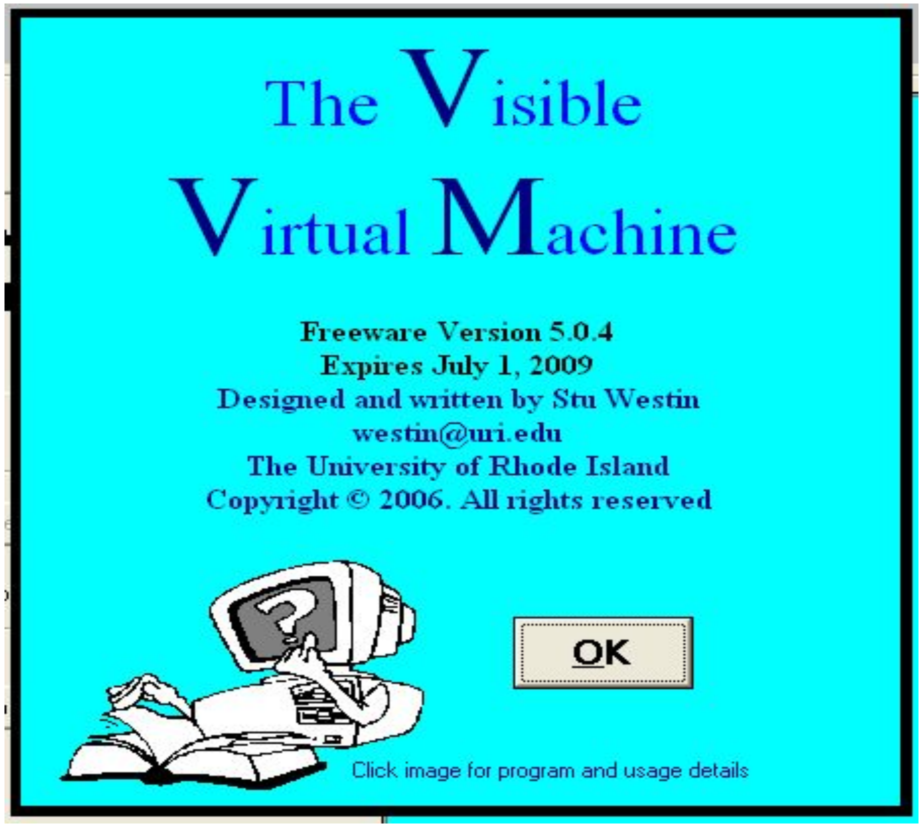
- **Any data operands required to execute the instruction are fetched from memory and placed into registers within the CPU.**



- **The ALU executes the instruction and places results in registers or memory.**



A virtual processor for understanding instruction cycle



The Visible Virtual Machine

Freeware Version 5.0.4
Expires July 1, 2009
Designed and written by Stu Westin
westin@uri.edu
The University of Rhode Island
Copyright © 2006. All rights reserved

The VVM Machine

- The Visible Virtual Machine (VVM) is based on a model of a simple computer device called the Little Man Computer which was originally developed by Stuart Madnick in 1965, and revised in 1979.
- The VVM is a <u>virtual</u> machine because it only appears to be a functioning hardware device.
- In reality, the VVM "hardware" is created through a software simulation. One important simplifying feature of this machine is that it works in decimal rather than in the traditional binary number system.
- Also, the VVM works with only one form of data - decimal integers.

Hardware Components of VVM

- **I/O Log**. This represents the system console which shows the details of relevant events in the execution of the program. Examples of events are the program begins, the program aborts, or input or output is generated.

- **Accumulator Register** (Accum). This register holds the values used in arithmetic and logical computations. It also serves as a buffer between input/output and memory. Legitimate values are any integer between -999 and +999. Values outside of this range will cause a fatal VVM Machine error. Non integer values are converted to integers before being loaded into the register.

- **Instruction Cycle Display**. This shows the number of instructions that have been executed since the current program execution began.

Hardware Components of VVM

- **Instruction Register** (Instr. Reg.). This register holds the next instruction to be executed. The register is divided into two parts: a one-digit *operation code*, and a two digit *operand*. The Assembly Language mnemonic code for the operation code is displayed below the register.

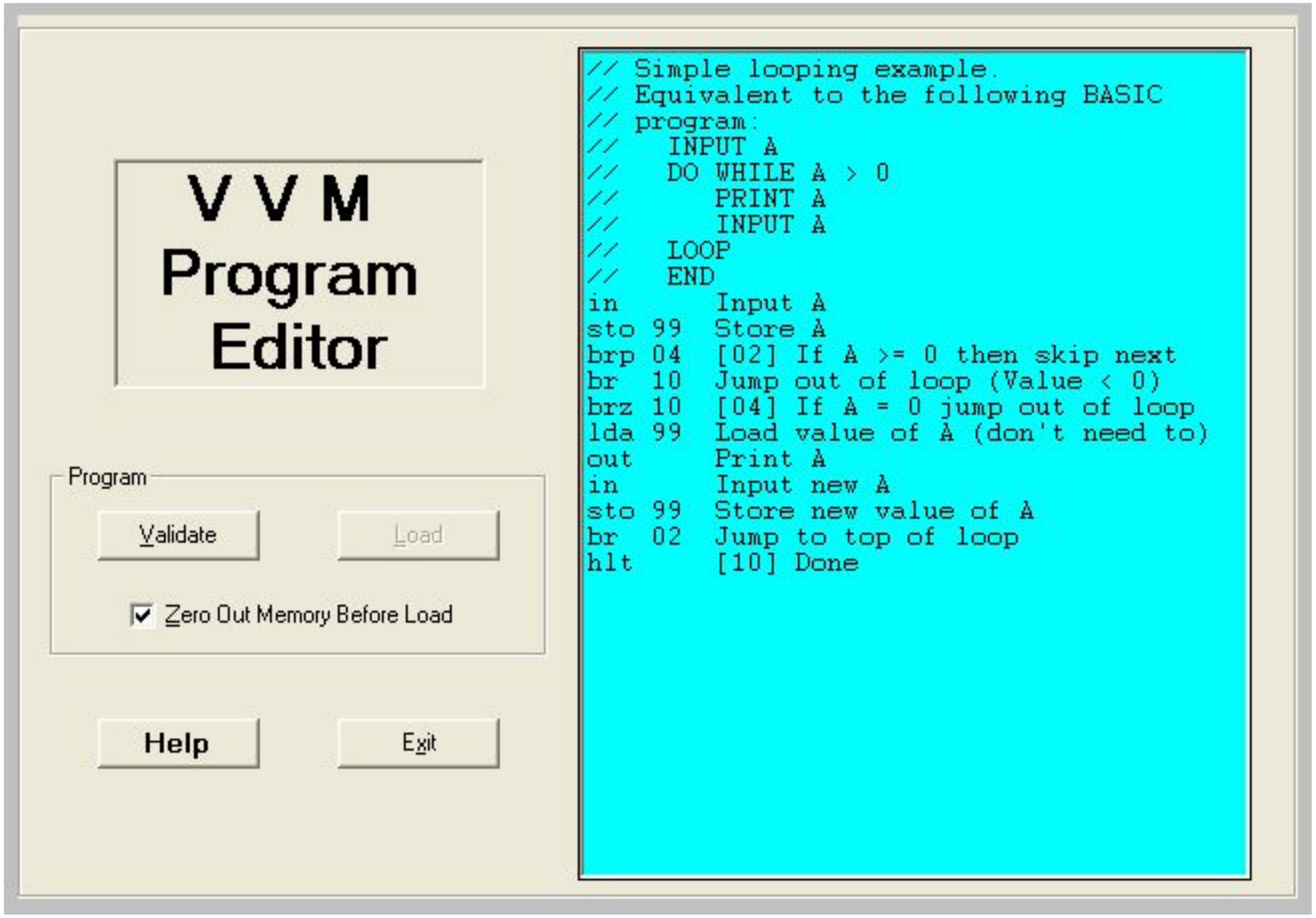
- **Program Counter Register** (Prog. Ctr.). The two-digit integer value in this register "points" to the next instruction to be fetched from RAM. Most instructions increment this register during the *execute* phase of the instruction cycle. Legitimate values range from 00 to 99. A value beyond this range causes a fatal VVM Machine error.

- **RAM**. The 100 *data-word* Random Access Storage is shown as a matrix of ten rows and ten columns. The two-digit memory addresses increase sequentially across the rows and run from 00 to 99. Each storage location can hold a three-digit integer value between -999 and +999.

Data and Addresses

- All data and address values are maintained as decimal integers.
- The 100 data-word memory is addresses with two-digit addressed in the range 00-99.
- Each memory location holds one data-word which is a decimal integer in the range -999 - +999.
- Data values beyond this range cause a data overflow condition and trigger a VVM system error.

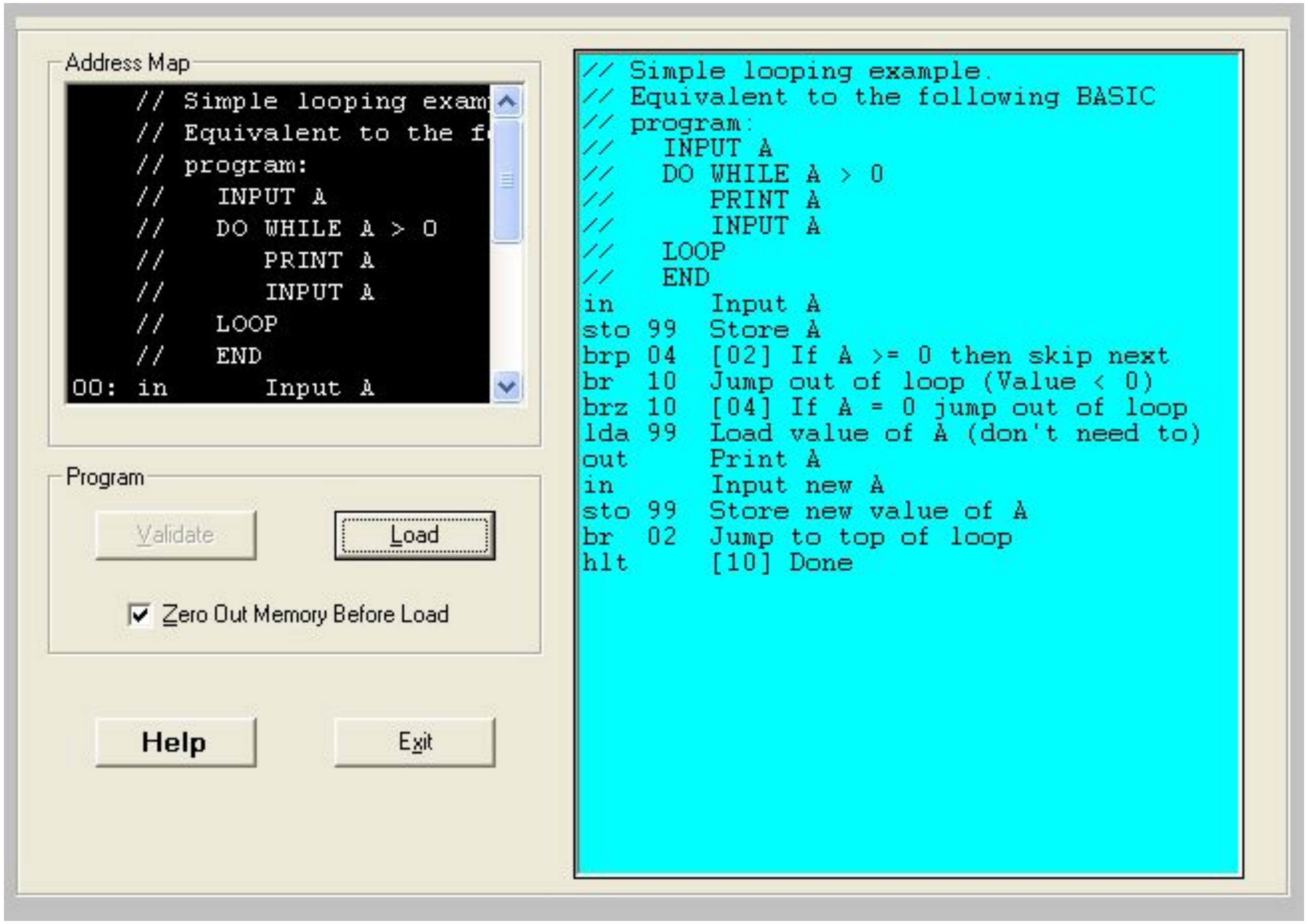
VVM Program Editor



[✓] Zero Out Memory Before Load

```
// Simple looping example.
// Equivalent to the following BASIC
// program:
//    INPUT A
//    DO WHILE A > 0
//       PRINT A
//       INPUT A
//    LOOP
//    END
in       Input A
sto 99   Store A
brp 04   [02] If A >= 0 then skip next
br  10   Jump out of loop (Value < 0)
brz 10   [04] If A = 0 jump out of loop
lda 99   Load value of A (don't need to)
out      Print A
in       Input new A
sto 99   Store new value of A
br  02   Jump to top of loop
hlt      [10] Done
```

VVM Program Editor

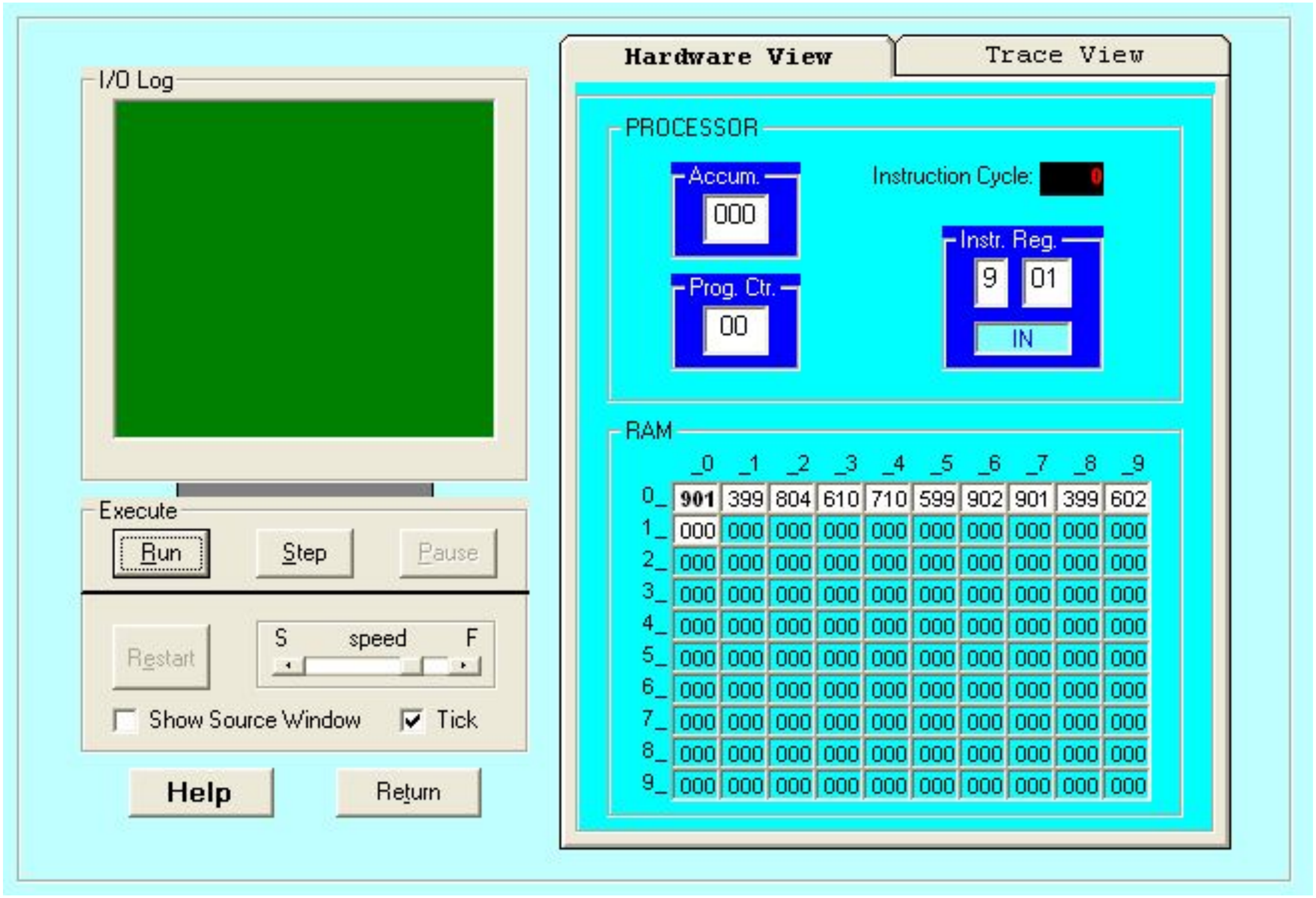


```
    // Simple looping exam
    // Equivalent to the f
    // program:
    //   INPUT A
    //   DO WHILE A > 0
    //      PRINT A
    //      INPUT A
    //   LOOP
    //   END
00: in      Input A
```

☑ Zero Out Memory Before Load

```
// Simple looping example.
// Equivalent to the following BASIC
// program:
//   INPUT A
//   DO WHILE A > 0
//      PRINT A
//      INPUT A
//   LOOP
//   END
in      Input A
sto 99  Store A
brp 04  [02] If A >= 0 then skip next
br  10  Jump out of loop (Value < 0)
brz 10  [04] If A = 0 jump out of loop
lda 99  Load value of A (don't need to)
out     Print A
in      Input new A
sto 99  Store new value of A
br  02  Jump to top of loop
hlt     [10] Done
```

VVM Structure and User Interface



PROCESSOR

Accum. 000

Prog. Ctr. 00

Instruction Cycle: 0

Instr. Reg. 9 01 IN

RAM

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9
0_	**901**	399	804	610	710	599	902	901	399	602
1_	000	000	000	000	000	000	000	000	000	000
2_	000	000	000	000	000	000	000	000	000	000
3_	000	000	000	000	000	000	000	000	000	000
4_	000	000	000	000	000	000	000	000	000	000
5_	000	000	000	000	000	000	000	000	000	000
6_	000	000	000	000	000	000	000	000	000	000
7_	000	000	000	000	000	000	000	000	000	000
8_	000	000	000	000	000	000	000	000	000	000
9_	000	000	000	000	000	000	000	000	000	000

VVM System Errors

- **Data value out of range**. This condition occurs when a data value exceeds the legitimate range -999 - +999. The condition will be detected while the data resides in the *Accumulator Register*. Probable causes are an improper addition or subtraction operation, or invalid user input.

- **Undefined instruction**. This occurs when the machine attempts to execute a three-digit value in the *Instruction Register* which can not be interpreted as a valid instruction code. See the help topic "VVM Language" for valid instruction codes and their meaning. Probable causes of this error are attempting to use a data value as an instruction, an improper *Branch* instruction, or failure to provide a *Halt* instruction in your program.

- **Program counter out of range**. This occurs when the Program Counter Register is incremented beyond the limit of 99. The likely cause is failure to include a *Halt* instruction in your program, or a branch to a high memory address.

- **User cancel**. The user pressed the "Cancel" button during an *Input* or *Output* operation.

The Language Instructions

- **Load Accumulator (5*nn*) [LDA *nn*]** The content of RAM address *nn* is copied to the Accumulator Register, replacing the current content of the register. The content of RAM address *nn* remains unchanged. The Program Counter Register is incremented by one.

- **Store Accumulator (3*nn*) [STO *nn*]** or **[STA *nn*]** The content of the Accumulator Register is copied to RAM address *nn*, replacing the current content of the address. The content of the Accumulator Register remains unchanged. The Program Counter Register is incremented by one.

- **Add (1*nn*) [ADD *nn*]** The content of RAM address *nn* is added to the content of the Accumulator Register, replacing the current content of the register. The content of RAM address *nn* remains unchanged. The Program Counter Register is incremented by one.

The Language Instructions

- **Subtract (2*nn*) [SUB *nn*]** The content of RAM address *nn* is subtracted from the content of the Accumulator Register, replacing the current content of the register. The content of RAM address *nn* remains unchanged. The Program Counter Register is incremented by one.

- **Input (901) [IN]** or **[INP]** A value input by the user is stored in the Accumulator Register, replacing the current content of the register. Note that the two-digit operand does not represent an address in this instruction, but rather specifies the particulars of the I/O operation (see Output). The operand value can be omitted in the Assembly Language format. The Program Counter Register is incremented by one with this instruction.

- **Output (902) [OUT]** or **[PRN]** The content of the Accumulator Register is output to the user. The current content of the register remains unchanged. Note that the two-digit operand does not represent an address in this instruction, but rather specifies the particulars of the I/O operation (see Input). The operand value can be omitted in the Assembly Language format. The Program Counter Register is incremented by one with this instruction.

The Language Instructions

- **Branch if Zero (7*nn*) [BRZ *nn*]** This is a conditional branch instruction. If the value in the Accumulator Register is zero, then the current value of the Program Counter Register is replaced by the operand value *nn* (the result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address). Otherwise (Accumulator >< 0), the Program Counter Register is incremented by one (thus the next instruction to be executed will be taken from the next sequential address).

- **Branch if Positive or Zero (8*nn*) [BRP *nn*]** This is a conditional branch instruction. If the value in the Accumulator Register is nonnegative (i.e., >= 0), then the current value of the Program Counter Register is replaced by the operand value *nn* (the result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address). Otherwise (Accumulator < 0), the Program Counter Register is incremented by one (thus the next instruction to be executed will be taken from the next sequential address).

- **Branch (6*nn*) [BR *nn*]** or**[BRU *nn*]** or **[JMP *nn*]** This is an unconditional branch instruction. The current value of the Program Counter Register is replaced by the operand value *nn*. The result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address. The value of the Program Counter Register is not incremented with this instruction.

The Language Instructions

- **No Operation (4*nn*) [NOP]** or **[NUL]** This instruction does nothing other than increment the Program Counter Register by one. The operand value *nn* is ignored in this instruction and can be omitted in the Assembly Language format. (This instruction is unique to the VVM and is not part of the Little Man Model.)

- **Halt (0*nn*) [HLT]** or **[COB]** Program execution is terminated. The operand value *nn* is ignored in this instruction and can be omitted in the Assembly Language format.

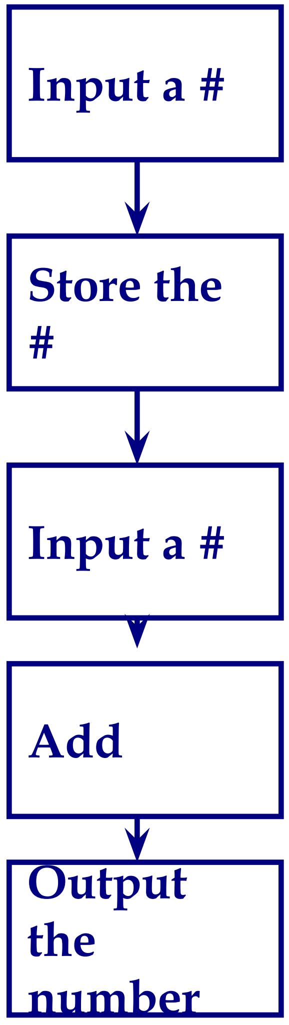
Embedding Data in Programs

- Data values used by a program can be loaded into memory along with the program. In Machine or Assembly Language form simply use the format "*snnn*" where *s* is an optional sign, and *nnn* is the three-digit data value. In Assembly Language, you can specify "DAT *snnn*" for clarity.

Assembly Language

- Specific to a CPU
- 1 to 1 correspondence between assembly language instruction and binary (machine) language instruction
- *Mnemonics* (short character sequence) represent instructions
- Used when programmer needs precise control over hardware, e.g., device drivers

Example - Add 2 Numbers



- Assume data is stored in mailbox address 90
- Let us write instructions (using Mnemonics)

```
00   IN;          input 1st Number
01   STO 90;          store data in memory
                location 90
02   IN;          input 2nd Number
03   ADD 90;          add 1st # to 2nd #
04   OUT;             output result
05   COB;             stop
90   DAT 00;          data
```

Example - Add 2 Numbers (Using Machine code)

Mailbox	Code	Instruction Description
00	901	;input 1st Number
01	390	;store data in memory location 90
02	901	;input 2nd Number
03	190	;add 1st # to 2nd #
04	902	;output result
05	000	;stop
90	000	;data

Program Control

- Branching (executing an instruction out of sequence)
  - Changes the address in the counter
- Halt

	Content	
	Op Code	Operand (address)
BR (Jump)	6	xx
BRZ (Branch on 0)	7	xx
BRP (Branch on +)	8	xx
COB (stop)	0	(ignore)

Instruction Set

ADD	1xx	ADD
SUB	2xx	SUB
STA or STO	3xx	STORE
LDA	5xx	LOAD
BR or BRU or JMP	6xx	JUMP
BRZ	7xx	BRANC ON 0
BRP	8xx	BRANCH ON +
IN or INP	901	INPUT
OUT or PRN	902	OUTPUT
NOP or NUL	400	No Operation
HLT or COB	000	HALT (coffee break)

Example - Find Positive Difference of 2 Numbers

00	IN	901	;input 1st Number
01	STO 10	310	;store data in memory location 10
02	IN	901	;input 2nd Number
03	STO 11	311	;store data in memory location 11
04	SUB 10	210	;subtract 1st # from the 2nd #
05	BRP 08	808	;test
06	LDA 10	510	;if negative, reverse order
07	SUB 11	211	;subtract 2nd # from the 1st #
08	OUT	902	;print result and
09	HLT	000	;stop
10	DAT 00	000	;used for data
11	DAT 00	000	;used for data